Software Engineering Education: Towards Ethical, Reliable, and Beautiful Software

Aikya Inuganti
University of Maryland Baltimore County
ainugan1@umbc.edu

Madhuri Goyal
University of Maryland Baltimore County
mgoyal1@umbc.edu

Mohammad Samarah†
University of Maryland Baltimore County
msamarah@umbc.edu

ABSTRACT

In this paper, we present our experience with an innovative pedagogical approach to software engineering in a graduate-level advanced software engineering course. Our approach to software engineering and software design education relies on six dimensions: 1) restating the goal of software engineering education to say that software must be conceived of, architected, designed, developed, deployed, maintained, and managed to be ethical, reliable, and beautiful; 2) software should be engineered as a service; 3) apply proven architectural principles; 4) use sound design principles; 5) create rapid multi-modal prototyping; and 6) bring the course learning objectives together by creating a term-long project that creates a solution to a real-world problem using an iterative process. The results from students' feedback have been very positive with students citing the benefits of the course particularly a) the realignment of software engineering education goals centered on creating ethical, reliable, and beautiful software, b) the focus on clean, sound, and efficient architectures, and c) blending of IEEE SWEBOK, modern microservice architectures, and emerging approaches from software engineering research and open source. We plan to continue developing the course and enhance it in the areas of software reuse, software product design, AI and software design, design for diverse users, and design for sustainability.

CCS CONCEPTS

•Software and its engineering~Software creation and management~Software development process management~Software development methods •Software and its engineering~Software organization and properties~Extra-functional properties~Software reliability **•Software and its engineering~Software creation and management~Designing software~Software design engineering** •Social and professional topics~Professional topics~Computing education~Computing education programs~Software engineering education •Applied computing~Education~Collaborative learning •Software and its engineering~Software creation and management~Software development techniques~Software prototyping •Software and its engineering~Software creation and management~Collaboration in software development~Programming

Designing '24, April 15–14, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0563-2/24/04.
https://doi.org/10.1145/3643660.3643950

KEYWORDS

Software Engineering Education, Software Engineering Graduate Programs, Software Design, Software Ethics.

Background

The course development was influenced by related works in software engineering education research and previous experiences in industry developing both commercial and bespoke products: including productivity applications, imaging, embedded software for consumer electronics, and high-speed software subsystems for storage appliances. In this section, we describe some of the related works.

In "A Brief Survey of Software Architecture Concepts and Service-Oriented Architecture," (Valipour et al., 2009) the authors explore software architecture complexities and introduce service-oriented architecture (SOA) as an effective framework for web software development. SOA was a major driver of improved software applications in the past two decades and the lessons learned from SOA is the foundation for software as a service and a cornerstone of the design of our course.

In "SOLID Principles in Software Architecture and Introduction to RESM Concept in OOP" (Madasu et al., 2015), the authors describe how design principles affect key aspects such as reusability, extensibility, simplicity, and maintainability, particularly in Object-Oriented Programming (OOP). The integration of these design principles builds a crucial bridge between theory and real-world applications. SOLID is adopted into the educational approach of this course while combining it with other design principles.

In "Introduction to the Special Issue on Software Architecture,"(Garlan & Perry, 1995) the authors provide a comprehensive view of software architecture with a 25-year perspective that is still relevant today. The paper provides a historical context of software architectures and a foundational viewpoint. Additionally in the paper "A Survey on Software Architecture Analysis Methods" (Dobrica & Niemela, 2002) the authors introduce evaluation techniques that are still relevant. It is important as students learn new design approaches to have a solid grounding of the historical context of related and differing approaches. The takeaway of these two papers into our approach is that historical context should influence and guide future design approaches. Addressing challenges in education, "Software Engineering Education: Challenges and Perspectives" (Ouhbi & Pombo, 2020) and "Software Engineering Education in the New World: What Needs to Change?" (Bass, 2016) advocate for innovative teaching methodologies. Our approach is similar in that

we were not satisfied with traditional software engineering course design and sought to improve it by using new methods. These works highlight the challenges in software engineering education and advocate for innovative teaching methodologies while considering industry practices. "C4 Skills in the Engineering Graduate" (Gupta & Gupta, 2023) adds a market-driven perspective, emphasizing design-based learning and skills alignment. This is a fundamental aspect of our course design that it needs to be focused on the design and building of software applications and software intensive products. "Ethics Is a Software Design Concern," (Ozkaya, 2019) brings a critical dimension to our approach by highlighting ethics as a fundamental design constraint. This paper predates the global pandemic and the recent introduction of AI tools to the public. Ethical considerations are even more critical now as we see AI tools being used throughout the software engineering process. "The UI Design Process," (McInerney & Sobiesiak, 2000) acknowledge the prevalent challenges in UI design, emphasizing the need for structured processes, effective communication, and collaboration. In our approach, we identified beauty is one of the three critical aspects of software both in external appearance and internal workings.

Expanding on the insights provided by the previously mentioned papers, in the "Academic Education of Software Engineering Practices" (Stettina et al., 2013) bring a valuable perspective on the delicate balance between hands-on activities and academic reflection in teaching software engineering. Our approach places great emphasis on practical learning and fostering a dynamic educational environment. The paper's emphasis on intensive coaching and agile practices is part of our approach in providing a comprehensive and effective educational experience.

"Closing the Gap Between Software Engineering Education and Industrial Needs" (Garousi et al., 2020) significantly addresses the real-world challenges faced by students entering the software industry. Our approach provides educators and practitioners educational curricula that stays ahead of industry by adapting educational approaches to meet the evolving needs of the software sector and a new category of software intensive products.

Exploring the ethical dimensions, "Ethics in Information Technology and Software Use" (Calluzzo & Cante, 2004) reaffirm our consideration of ethics as a fundamental design constraint in software engineering. We applied findings from the paper's review of students' attitudes and perceptions regarding ethical behaviors into our approach ensuring that our educational framework not only imparts technical knowledge but also instils a strong ethical foundation.

Taking lessons learned from SOA, SOLID principles, previous and current software design and architectures, ethics as a design constraint, and UI design, we applied these lessons in the context of software engineering education challenges, identifying critical attributes needed in software products today while considering the special needs of the software engineering practice and the constantly changing demands of the industry.

1 Introduction

In this paper, we describe an innovative pedagogical approach for an advanced software engineering course. Our approach consists of several dimensions and for each one we employ one or more proven software engineering methods. There are six dimensions: 1) restating the overarching goals of software applications and systems, 2) employing engineering software as a service, 3) using multiple prototyping modalities, 4) applying proven architecture principles, 5) implementing best practice design principles, and 6) building a term-long project.

The course blends content from a recent textbook on engineering software as a service using the Ruby programming language and the Ruby on Rails framework (Fox & Patterson, 2021), the IEEE Software Engineering Body of Knowledge v3 (Bourque et al., 1999), and relevant content from emerging software engineering research, open source, and commercial software. Topical coverage varies from topic to topic, some material is review of foundational topics, some is to establish background and context, and some is given deep coverage.

The first dimension is to restate the goals of software engineering and software design by emphasizing three attributes. We recognize that there are many desirable attributes and state that the most important ones are ethical, reliable, and beautiful. Therefore, software must be conceived of, architected, designed, developed, deployed, maintained, and managed to be ethical, reliable, and beautiful. This is more critical now given the rise of software-intensive products where software is the main component or the main product differentiation. Ethical means that all architectural, design, and construction decisions are made with the user's privacy and rights as a priority and that the software is for the service of the user first and foremost. And it uses the available computing resources responsibly. Reliable means that the software is safe to use, and protects the user from accidental mistakes, errors, and failures. It also means that it works with constraints in the user environment and fails safely. Beautiful means that the software is easy to use, pleasing to look at, easy to maintain, and embodies users' diversity in a meaningful way. Beauty in this context is not about style or color but rather a reflection of the beauty and diversity of its users among itself.

The second dimension is to engineer software as a service by creating independent, micro, and highly efficient services that can be combined together to create higher-level functionality. Thus, significantly increasing engineering software with reuse and for reuse creating cohesive, independent, and efficient software components and libraries.

The third, fourth, and fifth dimensions apply proven architectural and sound design principles and rapid multi-modal prototyping to engineer software that achieves desired results. Prototyping is achieved with low-fidelity, high-fidelity, and code-based prototypes. The chief architectural principles are reuse, cohesion, decoupling, and aspects among others. The chief design modelling properties introduced are completeness, consistency, and correctness along with the 6S checklist based on SOLID and SOFA principles. The sixth dimension is to create a solution to a real-world problem, issue, question, or gap using an incremental and iterative approach using five checkpoints. In the following section, we describe the dimensions in more detail.

2 Reimagining Software Engineering Education

2.1 Towards Ethical, Reliable, and Beautiful Software:

In the first dimension, students are challenged to rethink and reimagine the most important attributes of software, software applications, and software-intensive products. Acknowledging that traditional approaches have served us well in the past but need to be reevaluated given the rise of software-intensive products and the complexity of today's software. The student is introduced to the concept that software must be conceived of, architected, designed, implemented, built, deployed, maintained, and managed with three attributes first and foremost: it must be Ethical, Reliable, and Beautiful. We define ethical software as software that puts the user first by adhering to the following seven attributes:

1. It only does what it says it does.
2. It protects the user's privacy.
3. It does not use the hardware or software to monitor the environment of the user in aggregate or individually.
4. It does not fingerprint their software ID without their consent (individual identifying information does not leave the device without explicit consent).
5. It doesn't use their data for profit without their consent in aggregate or individually.
6. It does not consume their computational resources without need or consent.
7. It consumes computational and energy resources in an energy-efficient and sustainable way.

By being reliable, it embodies the following four attributes: a) it performs well under poor conditions, b) adapts to demands in the user environment, c) protects the user from accidents and errors, and d) fails safely with the rights and benefits of the user first and foremost.

By being beautiful, it has the following five attributes: a) it embodies all users in its operations, actions, and diversity, b) it is pleasing to look at and interact with, c) is aware of user likes, dislikes, preferences, and culture without biases or prejudices, d) it evolves and adapts as the users do, and e) it is easy to maintain and is beautiful to look at and work with its architecture, design, and code implementation. Our definition of beautiful goes beyond UI and human centered design to include the underlying code, architecture, and logical and physical aspects of the software.

2.2 Engineering Software as a Service (ESaaS):

The second dimension introduces the concept of engineering "software as a service", as the most effective and modern approach for many software applications. Here, the student is introduced to four pillars that support this approach:

1. Software as a Service (SaaS) is a modern approach to delivering software on demand through a web browser, a native desktop application, a mobile app, a dedicated device, or a software-intensive product.
2. Micro-services are fundamental to most SaaS applications, and they form the basis for adaptable, efficient, and reliable applications.
3. Creating a complex adaptive software system requires having a clear understanding of the system's purpose, usage, and evolution.
4. Low-fidelity and high-fidelity prototypes can aid in creating highly useable, efficient, and successful applications. We will cover this in more detail in dimension five.

To realize the benefits of engineering software as a service, the following principles must be applied:

1. Minimizing Complexity: Building and writing simple and readable code.
2. Anticipating Change: Building extensible software, that allows enhancements, change, and evolution to a software product without disrupting the underlying structure.
3. Constructing for Verification: Building software in a way that faults, errors, and gaps can be readily found by not only the authors of the code but also a variety of users with varying backgrounds and skill levels including test engineers, QA engineers, and end users.
4. Constructing for Reuse: Building software by creating reusable software assets including software libraries, modules, and components going beyond the boundaries of a single project, product, product family, team, organizational unit, or organization.
5. Constructing with Reuse: Building software with the reuse of existing software assets from libraries provided by the programming language, development platform, organizational software library repository, external COTS components, or Open-Source libraries.

2.3 Software Architecture and D-design:

The third dimension is concerned with teaching students the principles and methods for creating sound architectures. In this dimension, we ask students to employ principles and best practices from the IEEE SWEBOK in seven areas including 1) software design activities, 2) software design types, 3) software design principles, 4) key crosscutting issues, 5) software architecture views, 6) general UI design principles, and 7) software design strategies and methods.

The software architecture and design activities fall into three main categories: decomposition and componentization, the definition of component interfaces, and component level details definition to enable implementation and construction. The software design types follow a similar approach having D-design concerned with mapping software into components, FP-design identifying common aspects of the software to enable a family of software products, and I-design mapping users' needs to software features and functions.

The design activities and types form the basis for introducing the seven software design principles of abstraction, coupling and cohesion, decomposition and modularization, encapsulation/information hiding, separation of interface and implementation, sufficiency, completeness, and primitiveness, and

separation of concerns. Once the design principles are covered, key crosscutting issues are introduced to ensure students can apply lessons learned from the seven design principles during detailed design activities and software construction. The issues include concurrency, control and handling of events, data persistence, distribution of components, errors and exception handling and fault tolerance, interaction and presentation, and security.

To communicate and visualize the software architecture, multiple architectural views are introduced including logical views satisfying the functional requirements, process views showing concurrency issues, physical views showing distribution issues, and development views showing how the design is broken down into implementation components and units with explicit dependencies representation.

UI design is an important aspect of the software architecture and design. To have an effective UI design, the student is introduced to the seven general UI design principles including learnability, user familiarity, consistency, minimal surprise, recoverability, user guidance, and user diversity.

Then all aspects of the architecture and design are brought together by learning strategies and methods that enable sound and effective outcomes. This includes general, common, and crosscutting design strategies and methods. We introduce general strategies of divide-and-conquer, stepwise refinement, top-down and bottom-up, strategies that use heuristics, patterns and pattern languages, and iterative and incremental approaches. In addition, common strategies of function-oriented or structured design, object-oriented design, data structure-centered design, and component-based design. The cross-cutting strategies introduced include aspect-oriented design and service-oriented architecture.

2.4 Software Design and the 6S:

In the fourth dimension, we challenge the students to think deeply about design models as communication vehicles that model the essentials, provide perspectives, and enable clear and effective communication. Here, we draw lessons from IEEE SWEBOK. We then introduce modelling properties that are present in all effective models, the set of conditions that must be met for a good design model, and the types of models. The modeling properties introduced are completeness, consistency, and correctness measuring the model degree of requirements implemented, lack of contradictions in statements, constraints, and functions, and the degrees of defects present. In addition, the conditions that must be met prior, after, persist before and after the execution of the function: are preconditions, postconditions, and invariants. Once the design fundamentals are covered, we introduced three types of design models including informational, behavioral, and structural models each with its focus. While the first focuses on data representation with data properties, attributes, relationships, constraints, and sets, the second focuses on functions and features and how they behave to form a state machine, and control logic flow, and data flow. The third type focuses on the physical and logical composition into components and components parts with classes, objects, components, and packages.

This forms the basis to introduce the 6S Checklist with emphasis on best practice design principles including Site, SOLID, SOFA, Smells, Style, and Sign-off. These practices blend proven principles, automated tools, code metrics, and style consistency with peer code review to enable higher-quality design and code implementation. Site for example can be argued to be the most important item on this list when paired with Sign-off. Oftentimes, components, modules or functions are misplaced or miscategorized which creates a chain of undesirable poor qualities in the software design and therefore ultimately in its implementation. Peer code review and sign-off lead to higher quality design and better code and can identify misclassification and de-categorization early on. The SOLID and SOFA principles augment and complement each other.

SOLID and SOFA are object-oriented class and method design principles that describe best practices of good class and method design. SOLID includes five principles: Single Responsibility Principle, Open/Closed Principle, Liskov Substitution Principle, Dependency Injection Principle, and Demeter Principle. SOFA includes four principles: Short, do One thing, with Few arguments, and with a single Abstraction level. Code metrics and smell detectors can find violations of SOLID and SOFA. Design smells detect violations of SOLID principles while code smells detect violations of SOFA principles.

2.5 Rapid and Multi-Modal Prototyping:

The fifth dimension is rapid multi-modal prototyping. Here, we introduce the students to three levels of prototypes including low-fidelity, high-fidelity, and code-based prototypes. With low-fidelity prototyping, we aim to quickly map high-level design concepts into tangible artifacts. This can be accomplished with a paper prototype or clickable wireframes. High-fidelity prototypes are used when requirements are well understood, and the product prototype can be tested with actual users. At least three items are critical in a high-fidelity prototype including a visual design that has detailed UI elements and a look and feel similar to that of the end product, real-life content, and high interactivity. The last type of prototyping uses code-based tools to produce a high-fidelity prototype in the final product development environment. Typically, this type of prototype is identical in look and feel to the final product, uses the same tools and UI elements, and allows users to fully test the product features and functions. Students are challenged to find ways to use low and high-fidelity prototypes to rapidly iterate and evolve their proposed solutions and to meet the requirements of their project sponsor. In some cases, the code-based prototype is also explored while acknowledging its limitations and drawbacks.

2.6 Incremental, Iterative, Real-world Project:

The sixth dimension is to challenge the students to create a solution to a real-world problem, issue, question, or gap with an active sponsor as an option. The term project has ten high-level requirements, must be completed in one semester, and must be done

incrementally and iteratively using five checkpoints. The project domain may include but is not limited to applications in healthcare, education, non-profit social work, personal fitness, and children's games for education and learning. In the first semester that this course was offered during the COVID-19 pandemic, one interesting project was a national vaccine registry (NVR) that enables users, healthcare providers, healthcare facilities, and local and national governments to access, share, and manage vaccine data with patient privacy and transparency offering ease of use, patient education, and actionable insights for individuals, healthcare providers, administrators, and policymakers to make informed decisions.

The project's high-level requirements include the following ten items:

1. Solves a real-world problem, issue, question, or gap.
2. In partnership with a university department, a community group, a local company, or a regional partner.
3. Engineered as software as a service using SaaS architecture.
4. Must be accessible through a public URL.
5. Designed, implemented, tested, and deployed through iterative, prototyping, and incremental processes.
6. Requirements, design documents, and all code artifacts are accessible through a source control system.
7. Team collaboration is done in Atlassian Jira or a similar SCM tool to produce incremental releases.
8. Accessible through a web browser and at least one other modality including an iPhone or Android mobile app, native Windows OS, Mac OS, or Unix/Linux OS desktop application, a tailor-made device, or an IoT device.
9. Uses two or more database engine types in its implementation, for example, relational DBMS and Document Store; relational DBMS and Key-Value Store; or relational DBMS and Graph DBMS.
10. The implementation uses two or more programming languages. For example, JavaScript and Java; JavaScript and Python; HTML/CSS, JavaScript, and Ruby; JavaScript, Objective C, and Scala; or JavaScript and Rust or Go.

The five project checkpoints are:

- Checkpoint 1A: High-level requirements and low-fidelity prototype.
- Checkpoint 1B: High-level design and high-fidelity prototype.
- Checkpoint 2: Alpha functionality and quality with known issues and defects, detailed components design, and a list of functions available. Examples from the NVR project included the following features: User sign/up and registration, geolocation, OCR, text extraction, sharing and content security, and SMS push notifications.
- Checkpoint 3: Beta functionality with stable features, detailed components design, and a list of features available. Examples from the NVR project include OCR and text extraction iteration #2 (I2), image storage, machine learning and attributes categorization, sharing and content security (I2), and SMS push notifications (I2).
- Final Submission: RC functionality with near ready-for-production use, detailed components design, and a list of features available. Examples from the NVR project included machine learning and attribute categorization (I2), visualization, image storage (I2), sharing (I3), and SMS push notifications (I3).

Experiences and Lessons Learned

This course is part of a master's program in software engineering tailored for students who are looking to enter the field as software engineering practitioners with five pathways: software architecture, data engineering, software engineering management, software engineering education, and software engineering consulting. The first three pathways are primary and the last two are secondary. The program is structured as a master's in professional studies (MPS) and not a traditional master's degree (MS). It requires 30 credits, 6 core courses, 4 electives, and culminates in a capstone project with an internal or external sponsor. Students have the option to take electives from software engineering, computer science, data science, human centered computing, and information systems. The program was soft launched in Spring 2022 and is now in its 5th semester with enrollment of 80 students. The student population is a blend of traditional students, professional software engineers seeking an advanced degree, and career changers from related fields. The enrollment in this course has been averaging 70% traditional and 30% non-traditional students.

The course had a positive impact on a diverse group of students based on students' feedback and evaluation. One significant lesson learned revolves around the course's emphasis on a holistic approach to software development, intertwining ethical considerations, reliability, and aesthetics. This resonated strongly with Student Type 1, a traditional computer science graduate, who found it eye-opening and appreciated the course's real-world relevance. Student Type 2, a career changer with limited experience, praised the course for its hands-on and incremental problem-solving approach, reflecting the second crucial lesson learned: the development of practical, real-world problem-solving skills. Lastly, the third lesson learned underscores the course's focus that building reliable systems not only enhances technical skills but also establishes the foundation for earning users' trust, a realization that impacted both traditional graduates and those with field experience. Collectively, these lessons highlight the course's effectiveness in providing a comprehensive understanding of software engineering principles and practices, bridging theory with practical application for students with varying backgrounds and career stages.

We also learned that finding a sponsor for the term project is a multi-fold multiplier to the project's effectiveness and success. Although it might be difficult for some students to find the right sponsor, once they are on board and engaged, their positive impact on the project is felt almost immediately. Another lesson is the need to both update the content and tailor it to the students on a semester-by-semester basis. The updates and the tailoring levels varied from minimal to moderate. We found that the students are much more engaged and excited about the course when the content is up-to-date and feels very relevant. This is an area that we need to tune further to find the right point where the effort level is reasonable with the right impact level.

Conclusion

The results from the past three semesters have been very positive. Average enrollment was 18 students per semester. Students' feedback reflected an overwhelmingly positive outcome with an average course evaluation of 4.4/5.0 and a learning overall evaluation of 4.6/5.0. Students written response also showed a high level of satisfaction. Students at different stages in learning and experience mentioned that they benefited greatly from the course specifically in the course emphasis on a) realignment of software engineering education goals towards creating ethical, reliable, and beautiful software applications, software systems, and software-intensive products, b) the focus on clean, sound, and efficient micro and macro architectures, and c) blending in the right levels of teachings from the IEEE SWEBOK, modern microservices architectures, and emerging approaches from the field of software engineering research and mainstream open source software.

We envision that the course design will evolve more as we enter into a new period where AI tools and applications are used in meaningful ways in software engineering education both at the entry and advanced levels. We plan to collect additional data that would be helpful to publish a follow-on paper next year. Our plan for the next academic year is to tune the course in the following areas: a) Topical coverage and project scope diversity, b) Higher levels of engineering software with reuse and for reuse, c) Designing software-intensive products, d) Special considerations for designing with AI models, e) Designing for diverse users, and f) Designing for sustainability.

Acknowledgements

The authors are grateful for the support of the College of Engineering and IT, the Information Systems Department, and the Software Engineering Graduate Program. Authors Inuganti and Goyal are graduate students and TAs/RAs in the software engineering department and have taken this course. Author Dr. Samarah is the corresponding author. He envisioned and developed the course and taught it for the past 2 years. All authors contributed to the design and layout of the paper, the writing of the text, and the collection of relevant background and data. First two author names are ordered alphabetically.

References

[1] Valipour, M. H., AmirZafari, B., Maleki, K. N., & Daneshpour, N. (2009, August). A brief survey of software architecture concepts and service oriented architecture. In 2009 2nd IEEE International Conference on Computer Science and Information Technology (pp. 34-38). IEEE.

[2] Madasu, V. K., Venna, T. V. S. N., Eltaeib, T., Moalla, M. A., Almuslet, N. A., & Badaoui, A. (2015). Solid principles in software architecture and introduction to RESM concept in OOP, 2. 3159-40.

[3] Garlan, D., & Perry, D. E. (1995). Introduction to the special issue on software architecture. IEEE Trans. Software Eng., 21(4), 269-274.

[4] Dobrica, L., & Niemela, E. (2002). A survey on software architecture analysis methods. IEEE Transactions on Software Engineering, 28(7), 638-653.

[5] Ouhbi, S., & Pombo, N. (2020, April). Software engineering education: Challenges and perspectives. In 2020 IEEE Global Engineering Education Conference (EDUCON) (pp. 202-209). IEEE.

[6] Bass, M. (2016, April). Software Engineering Education in the New World: What Needs to Change? In 2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET) (pp. 213-221). IEEE.

[7] Gupta, C., & Gupta, V. (2023). C4 Skills in the Engineering Graduate: A Study to Align Software Engineering Education With Market-Driven Software Industry Needs. IEEE Transactions on Education

[8] Stettina, C. J., Zhou, Z., Bäck, T., & Katzy, B. (2013, May). Academic education of software engineering practices: towards planning and improving capstone courses based upon intensive coaching and team routines. In 2013 26th International Conference on Software Engineering Education and Training (CSEE&T) (pp. 169-178). IEEE

[9] Garousi, V., Giray, G., Tuzun, E., Catal, C., & Felderer, M. (2019). Closing the gap between software engineering education and industrial needs. IEEE software, 37(2), 68-77.

[10] McInerney, P., & Sobiesiak, R. (2000). The UI design process. ACM SIGCHI Bulletin, 32(1), 17-21.

[11] Ozkaya, I. (2019). Ethics is a software design concern. IEEE Software, 36(3), 4-8.

[12] Calluzzo, V. J., & Cante, C. J. (2004). Ethics in information technology and software use. Journal of Business Ethics, 51, 301-312.

[13] Bourque, P., Dupuis, R., Abran, A., Moore, J. W., & Tripp, L. (1999). The guide to the software engineering body of knowledge. IEEE Software, 16(6), 35-44.

[14] Bourque, P., & Fairley, R. E. (2014). SWEBOK v3. 0: Guide to the software engineering body of knowledge. IEEE Computer Society, 1-335.

[15] Fox, A., & Patterson, D. (2021). Engineering Software as a Service: An Agile Approach Using Cloud Computing (2nd ed., 2.0b7).