Informatics in Education, 2021, Vol. 20, No. 4, 515–532
© 2021 *Vilnius University, ETH Zürich*
DOI: 10.15388/infedu.2021.23

Teaching Software Engineering using Abstraction through Modeling

Mohsen DORODCHI[1], Nasrin DEHBOZORGI[2],
Mohammadali FALLAHIAN[1], Seyedamin POURIYEH[3]
[1]*Department of Computer Science, University of North Carolina Charlotte, USA*
[2]*Department of Software Engineering and Game Development, Kennesaw State University, USA*
[3]*Department Information Technology, Kennesaw State University, USA*
e-mail: mohsen.dorodchi@uncc.edu, dnasrin@kennesaw.edu, mfallahi@uncc.edu, spouriye@kennesaw.edu

Received: February 2021

Abstract. Teaching software engineering (SWE) as a core computer science course (ACM, 2013) is a challenging task. The challenge lies in the emphasis on what a large-scale software means, implementing teamwork, and teaching abstraction in software design while simultaneously engaging students into reasonable coding tasks. The abstraction of the system design is perhaps the most critical and theoretical part of the course and requires early engagement of the students with the necessary topics followed by implementation of the abstract model consistently. Normally, students would take such courses in the undergraduate curriculum sequence after data structures and/or object-oriented design/programming. Therefore, they would be able to learn about systematic modeling of software as a system. In this work, we address how to facilitate the teaching of SWE by introducing abstract modeling. Furthermore, functional decomposition is reviewed as a critical component which in turn, requires understanding of how different tasks are accomplished by enterprise software. Combining such pieces with concepts of architecture and design patterns of software provides foundational knowledge for students to be able to navigate around enterprise software in the real world.

Keywords: software engineering education, abstraction, modeling.

1. Introduction

The current state of software engineering (SWE) as a discipline has changed over the course of past 50 years due to many different reasons and it is still different from the traditional engineering disciplines which are built on solid theoretical foundations. One of the major concerns about SWE is the lack of agreeable structure within the discipline (Erdogmus, 2018) which seems to be always very dependent on the background and experience of the "engineer" who finds a solution to a given problem, rather than based on a common structured knowledge agreed upon the community (Erdogmus, 2018). Experts in the field argue that the roots of such a problem stands on the inherent multiplicity of dimensions of the real-world problems and the variety of potential solutions.

In educational domain, software engineering is offered as undergraduate and graduate level degree as well as a concentration within computer science or engineering. However, the educational pathway of SWE offers various approaches and models. As a core knowledge in the field of computing the joint task force of ACM/IEEE CS Curricula (ACM, 2013) recommended to include SWE as a set of required knowledge units aligned with the overall computing education into all computing disciplines and many schools are following this recommendation by practicing it. Yet, due to different factors indicated earlier, there are different approaches to both the content and offering of the SWE courses in different levels. An interesting model of teaching based on several different principles including 1) working on large code bases through open source, 2) modeling, and 3) dynamic project team experience through scaffolded and staged practices of active learning (Dorodchi, 2019).

In this work, we are expanding beyond a teaching model by offering a harmonized teaching paradigm in software engineering education, addressing the major myth with software engineering which believes it is not strongly backed up with theoretic background. The consensus is that SWE education is mostly focused on practices and is more dependent on the taste and common practices of the experts offering it. By overviewing the theory of computer science on abstraction and how it relates to SWE education, we attempt to show that such impressions are not necessarily true. Furthermore, our review reveals that there have been efforts in SWE field with the roots in the theory of abstraction. Furthermore, we present a case study around our idea of how to teach practice-based software engineering with abstraction.

This paper is organized as follows. The following section overviews the theory of abstraction in computer science followed by modeling and particularly visual modeling as it relates to system level design and development. Considering this discussion, we propose our visual representation of modeling as it relates to abstraction in SWE education. Design patterns as another abstraction tool for holistic system overview is presented in section 4 with a case study presenting a sample implementation based on the idea of this paper. We conclude our paper in the last section.

2. Abstraction in Software Engineering

Abstraction in the field of computer science theory and data structures is a very common and trivial discussion. However, in practice of software development, and requirement engineering, is spite of the efforts by many researchers, abstraction does not seem to be practiced as the key element of design and development of software. The notion of abstraction is an essential part of many scientific disciplines and has been integrated into the education of that discipline from different perspectives. For example, in mathematics, the pioneering work of Piaget on notion of reflective abstraction has been applied to teaching mathematical concepts in different levels to facilitate the cognitive process (Cetin, 2017).

In software engineering, Wang (Wang, 2007) established a thorough theoretical basis for SWE in which special attention has been paid to the theory of abstraction. Referring to the inadequacy of “abstractive and precise description means for software architectures and behaviors”, and the “perplexity of labor organization in large groups and large-scale projects” as the “primary technical deficiencies” for software engineers, students, and educators (Wang, 2007). After addressing the myths of “there is no theoretical foundation for software engineering” (Wang, 2007), the perception of the legendary computer scientists such as Von Neumann and Dijkstra about software as a stored programmed logic on computing hardware is discussed. Moreover, the author proposes a set of structured mathematical foundations through definitions and theorems to address the two main issues in software engineering: “1) how to design and implement a software system that one is not able to do by only oneself? and 2) how to cope with the development of a software system in which one does not completely know or understand the whole system and parts produced by other team members?” (Wang, 2007). The author discussed that such issues raise due to the inherent complexity of large-scale software (Wang, 2007). To address the aforementioned questions, Wang (2007) starts the theoretical foundations of software engineering by discussing on the philosophical foundations of software engineering which tries to connect the abstract world with the physical world through multiple levels of abstraction. However, for the most part the software as a solution is not always considered tangible. Based on this foundational viewpoint, a design intensive solution can be provided to any of the infinite possible problems that software engineering deals with.

Wang continues the discussion on all aspects of mathematical foundations of SWE, which similar to philosophical foundations, provides top-level abstraction means. Followed by the overview and outlining of computing foundations (e.g., modelling, automata and finite state machine, etc.), linguistic foundations (e.g., formal language theory, syntax and semantics of programming languages, etc.), and information science foundations (e.g., completeness, consistency, exactness, feasibility, verifiability, etc.) as the major theoretical foundations. Moreover, the interdisciplinary organizational foundations, and principles and perspective chapters are presented and discussed as summarized in Table 1.

Table 1
Theoretical foundations aspects of SWE from (Wang, 2007)

Theoretical Foundation	# Of Chapters
Basic foundations of SWE (Philosophical, Mathematical, Computing, Linguistic, Information Science Foundations of SWE)	5
Organizational Foundations of SWE (Engineering, Cognitive Informatics, System Science, Management Science, Economics, and Sociology Foundations of SWE)	6
Fundamental SWE principles and perspectives	3

As mentioned before, abstraction in the field of SWE has been discussed mainly around the topics of requirement analysis and specification (Gacitua, 2010) and design of the complex software systems (Medvidovic, 1996), (Van der Westhuizen, 2006), (Rugaber, 2006). An early work by Kiczales (Kiczales, 1991) discusses the paradox of abstraction in software engineering in hiding details while providing enough details for developers. The idea of metaobjects is presented to solve this issue in which abstraction provides details of implementations. Abstraction for reasoning is discussed in another early work in (Giunchiglia, 1992). Salzer (Salzer, 2010) discusses the idea of the Atomic Requirement Specifications (ATRS) as well as the importance of the notion of abstraction, abstraction level and its hierarchy in SWE in modeling the requirements.

3. Software Modeling

Traditional modeling in software engineering follows the structural designs of programs around the processes whereas the object-oriented model (or the derivatives of that such as service-oriented or aspect-oriented) are focusing on the concept of "object" which includes data, behavior, and actions. In this work, we focus only on the two fundamental models of structural analysis and object-oriented design as follows in the next two subsections with a discussion on a hybrid model to provide different conceptual views of the system considering different layers of abstraction.

3.1. *Structural Analysis and Modeling*

One of the main models used in structural analysis and modeling includes Data Flow Diagram (DFD) with the context diagram as the general model of abstraction. The decomposition diagram of the DFD's helps with the identification of the major modules and the essentials of each module as well as providing an overall view of the system and the tasks accomplished in each part of the system. Such a model helps in dividing the big pictures into a set of independent modules which facilitates the development by reducing the overall complexity. This concept and the role of modeling in developing an abstract view is depicted in Fig. 1. As shown the modular design through multiple DFDs allows the system developers to get a holistic view of the system under development.

3.2. *Unified Modeling Language (UML)*

Unified Modeling Language (UML) has drawn the attention of software engineering educators and many efforts have been made to facilitate the practical aspects of UML (Tuparov, 2007). In this section, we review UML in the context of abstraction in system level design and development. Furthermore, we briefly discuss how the UML diagrams relate to different layers of abstraction.

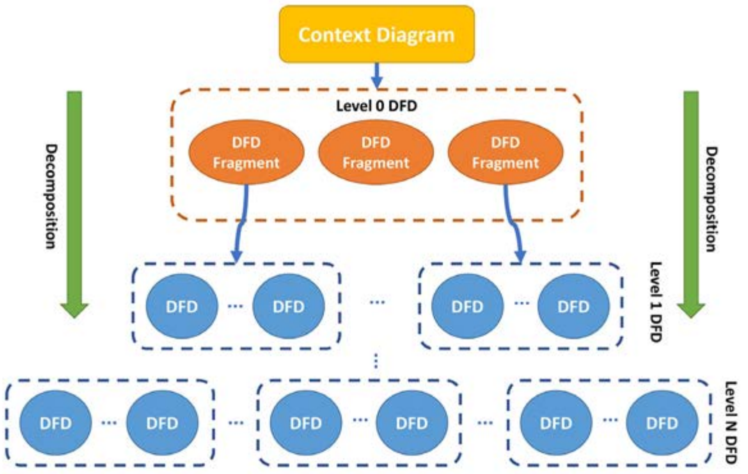

Fig. 1. Using structural analysis with DFD to visualize the entire system model.

3.2.1. *Brief Explanation of UML Diagrams*

UML diagrams represent two aspects of a system as shown in Fig. 2, the structural and the behavioral. Structural diagrams refer to the static system's features being modelled on different abstraction levels and represent the essential components of the system. The structural diagrams include class, object, component, package diagrams, compos-

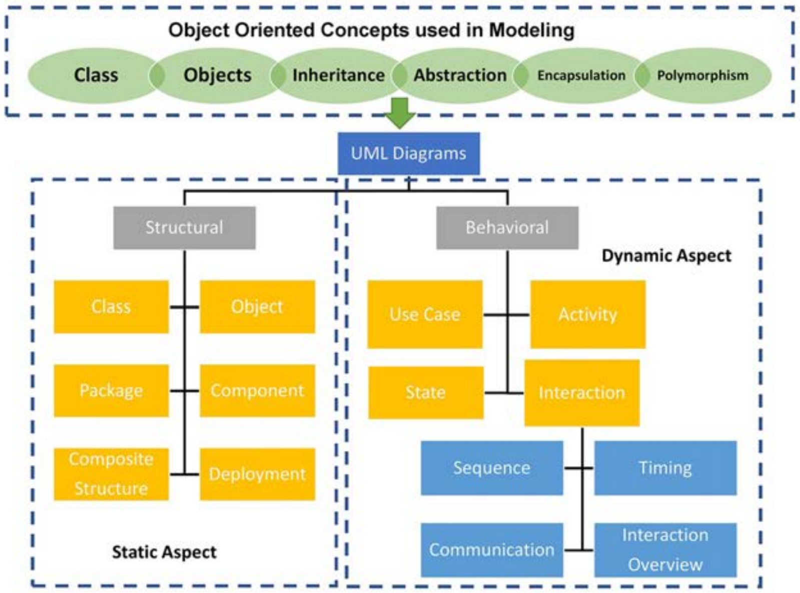

Fig. 2. Different types of UML diagrams.

ite structure, and the deployment. The class diagram represents the conceptual object-oriented model of the software system, including classes, attributes, methods, and the static relationships amongst classes whereas an object diagram is an instance of a class diagram and describes data structures at a particular instance. The component diagram breaks larger components of a system down into smaller components and illustrates the relationship between those components, and the package diagram shows dependencies between different packages in a software system. The composite structure diagram is generally used in modeling a system to represent how objects are composed at runtime, and the deployment diagram helps to model deploying of an Object-Oriented software system on the actual machines (Booch, 2017).

On the other hand, the behavioral or interaction diagrams focus on how the system's behavior changes and the objects interact while the system is running. There are four main types of diagrams in this category. The use case diagram models how users interact with the system and shows how a system provides users' needs. The actual use cases are normally modelled separately via textual models such as casual or fully dressed use cases. An activity diagram illustrates how system components work simultaneously and in parallel to help with visualizing workflow of actions taken in use cases. In object-oriented approaches, we are using state machine diagrams for a class to show the lifecycle of an object. State diagrams are mostly used to describe the behavior of an object across several use cases. Lastly, the interaction diagram includes four types of diagrams, namely sequence diagram, communication diagram, timing diagram, and interaction overview diagram. Interaction diagrams describe the behavior of several objects in a single use case and show a sequence of activities for those objects (Booch, 2017). It is worth noting that there have been efforts in promoting modeling and particularly UML, by the open-source community to help with the design practice (Aldaeej, 2016) as well as for educational purposes (Dorodchi, 2019).

3.2.2. *Layers of Abstraction for Each of the UML Diagrams*

The level of abstraction of each of the UML diagrams represent the details level of the system. We have derived these levels based on the literature and known practices as shown in Fig. 3. In pure object-oriented programming one can use all these diagrams based on the provided levels of abstraction and flow of data from one object to another. Furthermore, we will discuss in the next section the hybrid design models based on levels of abstraction for the more complex enterprise systems. We propose using traditional structured modeling such as DFD, ERD, and so forth to help with a more thorough understanding of current enterprise systems built based on different programming paradigms, languages, and supporting technologies.

Object-oriented analysis and design method applies object-orientated concepts to analyze and design a software system using visual modeling in the development life cycle. In this approach, the use case diagram is the system-level unit for defining requirements (Stumpf, 2006) and describes the system's overall flow of events. That is an abstract model of the system from the user's perspective. Such a model can be easily communicated with the stakeholders, managers, and software developers. The use

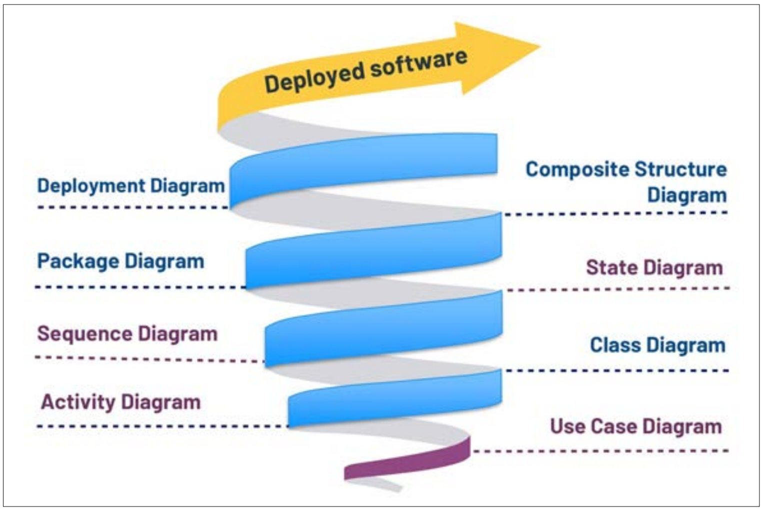

Fig. 3. Levels of abstraction of UML diagram helping with Object-Oriented Analysis and Design.

case diagram objects are utilized to create class diagrams as the building blocks of the software (Rumbaugh, 2010).

As the next level of abstraction, the activity diagram depicts how the activities of the particular use cases are navigated in a special order to perform the actual task and provide the required service. At the lowest layer of abstraction, the sequence diagram illustrates the order of the actual methods being called when a message is sent from the system user (Stumpf, 2006). Furthermore, the state diagram is to provide the dynamics of the behavior of the system through its methods (Rumbaugh, 2010).

Fig. 3 depicts all the above discussions. As shown, the developers start from abstract models all the way to more detailed models of the system and eventually the real software can be built and deployed.

4. Design Patterns in Software Engineering Education

Design patterns as a model of abstraction has been around in the fields of architecture (Alexander, 1977), education (Maher, 2020, Dorodchi, 2020), and SWE (Wedyan, 2020). Software engineering discipline has certain dimensions that are very important both for students in the educational domain and for the professionals practicing in the field. These include familiarity with key skills in software development, understanding major components of the enterprise systems, and interpersonal skills to work in high-stake teams.

In this section, we overview the applicating of design patterns in different disciplines and discuss how modeling abstraction by design patterns can help with software engineering education from two perspectives, one from educational standpoint to address the teamwork and the second to teach SWE architectural concepts to smoothen the implementation process for the students. These perspectives are discussed in detail in the following subsections.

Design patterns provide solutions to the common recurring problems that software developers face in developing high-quality software systems (Wedyan, 2020). Design patterns and pattern languages originated from Alexander *et al.* in the field of architecture in 1977 (Alexander, 1977). Alexander *et al.*'s intention was to democratize town planning by proposing a set of conceptual frameworks that could be applied by ordinary people in buildings and architecture (Dehbozorgi, 2017).

Later in the mid-1990s Gamma *et al.* (Gamma, 1995) proposed applying the concept of design patterns in the field of object-oriented software development (Wedyan, 2020). They proposed the gang of four (GoF) patterns which includes a set of 23 patterns in three categories of structural, creational, and behavioral patterns (Wedyan, 2020). The creational patterns break down into five patterns that address problems related to the creation of objects. The structural category includes seven patterns that provide the relationship between objects and the behavioral category consists of eleven patterns that help in designing how the objects interact with each other. Application of design patterns in software engineering has multiple advantages such as better decision making, enhancing communication of design decisions among developers, improving the software reusability, and as a result saving development cost, in addition to satisfying the non-functional requirements of the software systems (Wedyan, 2020).

Similar to the concept of design patterns in SWE that offer generic repeatable solutions to the known problems, pedagogical design patterns formalize successful practices that can address recurring problems that educators face in the educational setting (Dehbozorgi, 2018). Pedagogical design patterns do not provide absolute and finalized solution, however, they present tested and proven paradigms that can be implemented in educational settings in different ways (Dehbozorgi, 2018). This is well-aligned with the original definition of design patterns which states, a design pattern "… describes the core of the solution to the problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" (Alexander, 1977).

Pedagogical design patterns address diverse problems that instructors face in both lecture-based and active learning settings in different formats (Dehbozorgi, 2018). Most of the existing patterns are rooted in the original format of Alexander's pattern that includes 'problem' and 'solution' as the main components (Dehbozorgi, 2018). In a more recent study, the authors proposed an object-based pattern model which has different attributes to address problems specifically in collaborative student-centered learning. The multiple attributes of this pattern model allow capturing diverse dimensions of teamwork in software engineering classes (Dehbozorgi, 2018–2017). This model has four main constructs of 'pattern name', 'meta-data, 'pattern core', and 'implementation' (Dehbozorgi, 2018). The pattern core components include problem, solution, rationale, and pitfall. The solution is extended to the second level set of attributes that capture diverse variations of teamwork such as team formation, team size, duration of teamwork, etc. (Dehbozorgi, 2018).

In the next subsection, we propose using the developed object-based pattern model to implement and scaffold teamwork in SWE classes.

4.1. *Pedagogical Design Patterns for Software Engineering Education*

Design patterns for teaching have been used to offer models of teaching to instructors. The idea is to help instructors practicing active learning address problems they face in their teaching in such settings (Maher, 2020). Collaborative learning and high-stake project teams are always of the major interests of educators and there are challenges of implementing such teamwork in classrooms. In software engineering education, the project teams need to be addressed thoroughly and utilized throughout the course. One major issue has always been the fact that students do not collaborate or communicate very efficiently.

In this study, we propose a set of teamwork patterns as abstractions to scaffold collaboration in SWE courses. These patterns are divided into three groups of pre-class, in-class, and post-class patterns. Although each pattern is at a certain level of abstraction that can be implemented individually, they can be applied in parallel or sequentially as well. The patterns of each category are presented in Fig. 4.

Fig. 4 lists the pre-class and in-class teamwork patterns which are based on the patterns in (Maher, 2020). In particular, the "Teamwork Deliverables" pattern is presented in Table 2. The pre-class patterns are mainly applicable to the intro-level SWE classes in which students need to learn the key skills of software development before applying them in their team projects.

The in-class patterns can be applied at any course level depending on the importance of teamwork into students' evaluation. For example, the low-stake teams better suit the introductory level classes (CS1) in which the emphasize is on peer learning, practicing teamwork, and developing interpersonal skills amongst students. Such teams do not generate a collective major output as a product of the course, however, since social construction of knowledge is the key purpose of these teams, students practice how to socialize, collaborate, and learn. After students pass the beginners phase the mid-stake-teams can be applied in which there are some team artifacts that are considered in the team evaluation. Finally, we have the high-stake-teams pattern that is most applicable in capstone classes where students work together to produce a final artifact.

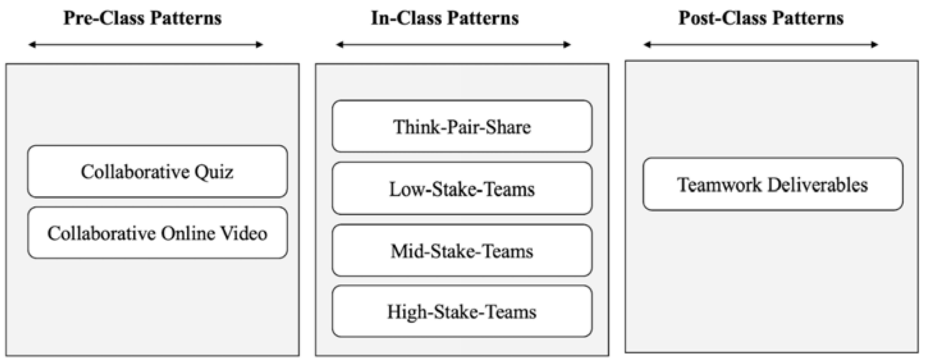

Fig. 4. The Teamwork Patterns (Dehbozorgi, 2020).

Table 2
Teamwork Deliverable Pattern

Teamwork Deliverables	
Metadata	
Pattern Focus	Learning/content delivery
Problem category	Collaboration/Performance
Implementation	Outside class
Pattern Core	
Problem	The class time doesn't allow students to finish scrums, the project deliverables and the presentation in the high-stake teams.
Solution	Provide clear instructions on what students need to prepare after class in the teams (this can include the roles assigned to the team members). Introduce some collaboration platforms that they can communicate with. Have a consistent plan for peer evaluation of team members for their collaboration outside class.
Rationale	Students have clear understanding of what is expected from them to do within a certain time frame. This helps students learn how to manage their time and plan properly to finish their assigned task in a disciplined manner.
Pitfall	Students may not take the teamwork outside class as seriously as in-class and validating the peer evaluation can be a challenge for the instructor.

The post-class patterns are mostly applicable to the higher-level courses in which students work together outside class setting to either finish a development phase (sprint) or work on the deliverables and presentations of the project to the instructor or the industrial partners. Furthermore, we consider and adopt an additional pattern that provides a generic solution for assessing teamwork. This pattern is named "*Teamwork Grade Assignment*" (Maher, 2020).

According to Alexander (Dehbozorgi, 2018), patterns get more values when their relationship is presented in the given domain which is called the pattern language. To show the dependencies or sequence of adopting the patterns we use concept map which represent our pattern language to adopt, adapt, and scaffold the teamwork in software engineering classes. Fig. 5 depicts the developed pattern language.

As discussed, design patterns can provide abstract solutions to the problems of implementing teamwork in software engineering classes. It is worth noting that, while these patterns provide cues to instructors to implement teamwork efficiently based on the best practices, they do not prescribe explicit design decisions and the educators have total freedom to apply them in a way that best fits their class settings. In the following subsection, we discuss adopting design patterns to deliver the architectural concepts to the students in the field of software engineering.

4.2. *Is there a Suitable Design Pattern to Teach?*

As noted in the pedagogical design patterns section, the work of Alexander (Alexander, 1977) has been the motivation for pattern development to support reuse of

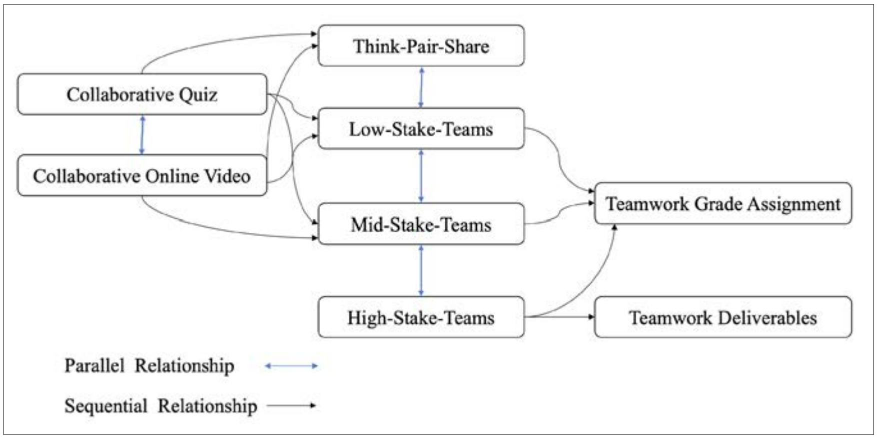

Fig. 5. Teamwork Patterns Concept Map.

knowledge, insights, components and providing domain-specific vocabularies to facilitate communication in project teams and to help organizing expert knowledge into a standardized format. Software patterns are also useful for pedagogical purposes and can help in understanding and evaluating existing software systems. Conceptualizing the big picture of system development requires experience and therefore, it is not a straightforward process for students. To achieve the holistic view of a system, design patterns seem to be a good candidate in addition to the modeling. However, there have been arguments against the usage of patterns due to its negative impact. In a study by Budgen, seven different patterns were studied, and it was concluded that design patterns can generate unwanted effects (Budgen, 2013). Inexperienced software developers may face challenges in applying patterns whereas experienced ones find it very effective.

We argue that these problems lie in the abstraction level of design patterns and lack of proper transitions between design layers to match the learners background and readiness to deal with the practical implementations of the patterns in real world. The complexity and speed of technology and development methodologies is another challenge for proper scaffolding of the general patterns.

Considering the above argument, we present a case study based on scaffolding the Model-View-Controller pattern (Reenskaug, 2003) to show how it can be adjusted and adapted to diverse environments and technologies. Similar efforts have been done such as the work in (Cortez, 2015) discussing how the extended MVC into virtual model-view-controller (Virtual MVC) simplifies the development of service-oriented architecture (SOA). In another work by Curry and Grace, inspired by the challenges in the design of the ubiquitous and pervasive computing systems, the idea of flexible system infrastructures as self-management system based on the extension of MVC is discussed (Curry, 2008). The proposed system can adapt to operational requirements and environmental conditions dynamic challenges.

Moreover, we believe that proper selection and application of design patterns help learners understand the big picture of developing software systems and realize how different components of the enterprise systems are related and interact with each other.

4.3. *Model-View-Controller (MVC)*

MVC was introduced by Trygve Reenskaug in 1978 to address, according to (Reenskaug, 2003) the “user’s mental model of the relevant information space”. In this way, the user was indeed able to further inspect and edit the design. Furthermore, it was extended to include current day to day challenges in iterative design and development (Reenskaug, 2003). Later, several different researchers and engineers implemented MVC in different contexts. For example, a group implemented MVC for the Smalltalk programming language in 1987 and eventually MVC was accepted as a general design concept in 1988. Nowadays, the MVC pattern is widely used in modern web application development.

The concept of the MVC pattern creates the foundation for several recent development frameworks such as Laravel (Rajput, 2020), Spring (“Spring Web MVC”, n.d.), and similar ones. Development frameworks are built to provide necessary tools and supporting libraries to hide some of the development complexities. The MVC asks developers to split a system into three main parts: The Model, the View, and the Controller. The model represents the entities or objects that are used to store application data. The view layer, on the other hand, is visually modeling the gateway to the system by users to make it easy to understand how the ‘model information’ is accessible and interacted with. Finally, the controller includes the main logic of the system and is placed between the model and view layers to facilitate communications and interactions between them. In addition, the controller plays a vital role in performing all the logic in the applications to ensure it is worked properly and is applied to execute the actions we can perform inside our system. In short, the MVC pattern makes building an application easier by separating these three layers.

When we construct an application using MVC paradigm, the dependencies between layers which create many issues during the development are isolated and reduced. Furthermore, it is easier to maintain a system built in MVC pattern and is notably scalable and expandable (due to reusable components) to the needs. In addition, all the tasks of the different system layers are clear and transparent to different developers. Fig. 6 depicts the MVC design pattern as an architecture diagram and shows how these layers are interacting with each other. The user through the view can notify the system what information is needed to be updated and the controller performs the task and updates the model. During the information retrieval, the model would notify the controller to provide the information to the user through the view. This is a very transparent model of an application.

We argue that MVC provides the right level of abstraction to be used for teaching and learning as it is built to visually model a complex system. Higher levels of abstraction of design patterns are not very usable by learners and practitioners as indicated in (Budgen,

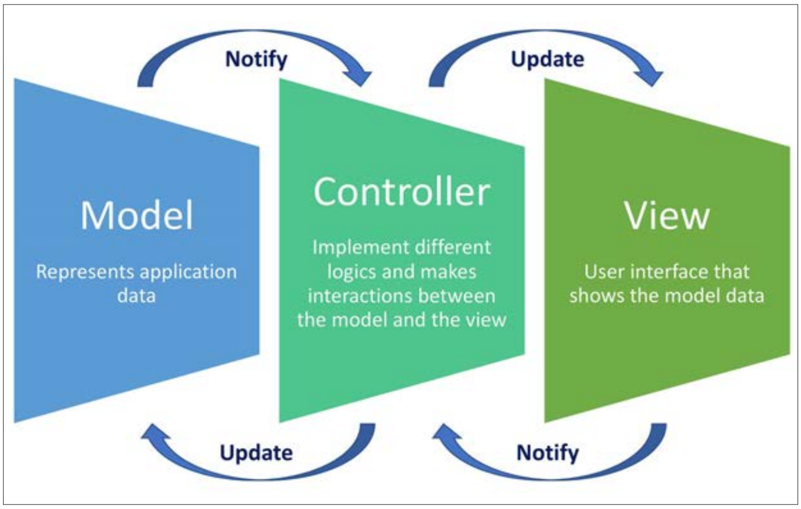
Notify
Update
Model
Represents application data
Controller
Implement different logics and makes interactions between the model and the view
View
User interface that shows the model data
Update
Notify

Fig. 6. MVC Architecture Diagram.

2013), however, a suitable pattern such as MVC with proper choice of technology can help the learners significantly as discussed in the next subsection.

4.3.1. *Case Study of Transforming Design Pattern to Practice*

As a case study we are going to represent an open-source framework which is specifically built for MVC-based web application development called Laravel. Web applications are built widely as internal and external special purpose enterprise software and evaluated by two major quality factors of stability and scalability. The two major threat to these factors are unstructured codes and architecture. Moreover, a project team needs to decide on assigning developer roles based on the enterprise application architecture and the developers' expertise. For example, frontend and backend developers need to understand the overall design clearly to be able to follow and guarantee the system's integrity and particularly the business rules (Pop, 2013). There are several frameworks that tackle this issue for system designers by offering the built-in design pattern and architecture.

One of the well-known web development frameworks is Laravel which follows MVC based architectural pattern and is fully object-oriented. Laravel allows system designers to model their system based on the built-in pattern. As a result, developers can create full-featured enterprise software based on the best practices and a built-in design pattern.

Fig. 7 illustrates a Laravel project structure. For each layer of the MVC pattern, there is a directory in the project as highlighted. Code development as well as teaching and learning of it follows the theory in an abstract form with a clear mapping to the actual code development in a smooth and straightforward way.

The design and implementation processes can be easily merged using such concepts. For instance, Fig. 8 represents a simple class diagram that a system designer can model

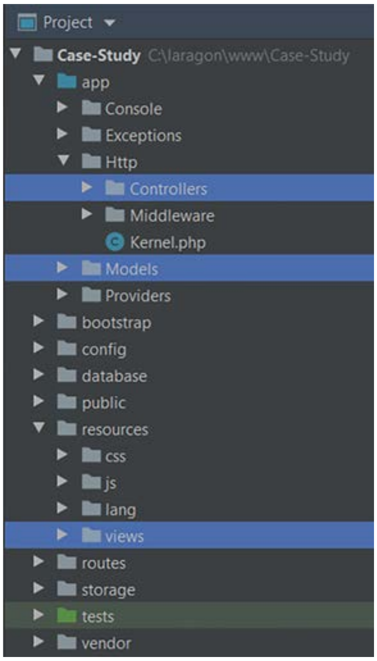

Fig. 7. Laravel MVC project structure.

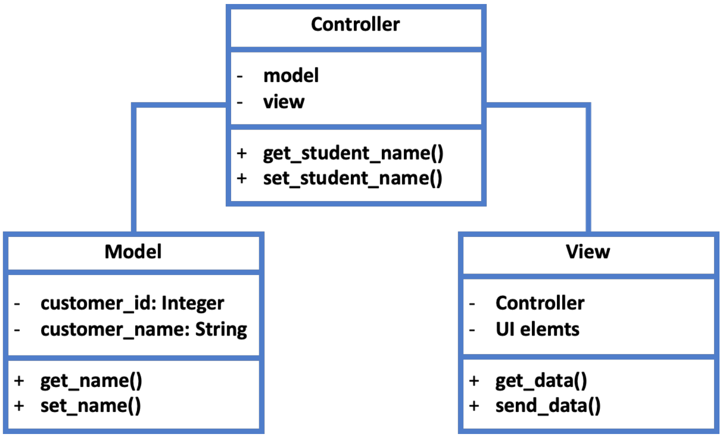

Fig. 8. Class diagram for implementing MVC pattern in Laravel.

after setting up a Laravel project. Laravel framework gives software system designers a logical and analytical insight to build integrated and robust modeling.

Furthermore, MVC frameworks enable the software system designer to focus on modeling without taking time to implement patterns and practices as it complies with the agile software development process. Fig. 9 shows the implementation of the MVC pattern in Laravel software architecture.

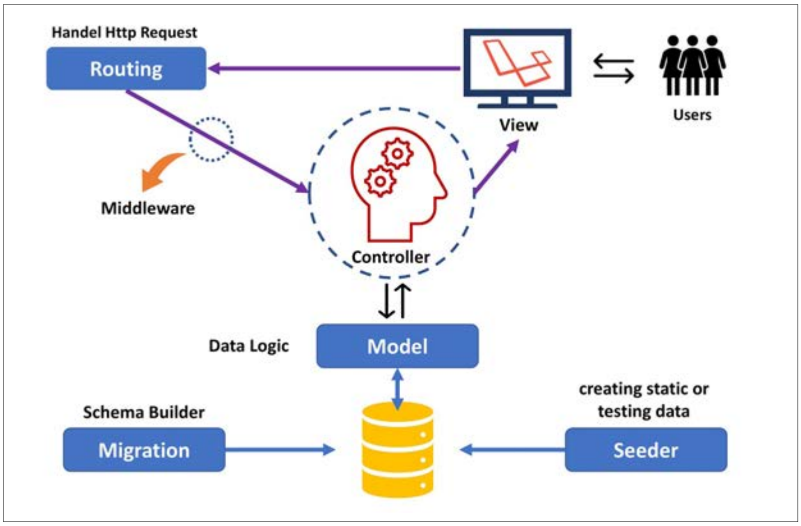

Fig. 9. Laravel framework architecture diagram.

Therefore, by engaging a modern framework such as Laravel, developers are reinforced to follow an abstract pattern such as MVC. The underlying robust tools integrated into the framework facilitates the reinforcement process to provide a truly ideal model for large enterprise application design (Rajput, 2020).

5. Conclusion

Abstraction as a fundamental notion in software engineering is presented in this work emphasizing on hierarchical representation to simultaneously hide and reveal the complexity of current information systems. This process facilitates the communication about the system within the development team and with the stakeholders.

In this study, we applied the concept of abstraction by discussing on modeling and adapting the notion of design patterns. Modeling provides the step-by-step approach in gaining the holistic view of a system while understanding what needs to be done. Based on abstraction, modeling techniques need to be used in a hierarchical order to help the learns. Hybrid modeling techniques such as structural and object-oriented can help with a multidimensional view of the system.

Moreover, we adapt the concept of design patterns to software engineering education in providing scaffolded models for collaboration and teamwork practices. We demonstrated examples of how to adopt and adapt object-based design patterns for such purposes.

And finally, we demonstrated how the practical knowledge of architectural patterns such as MVC are essential to help learns understand system development. Some traditional patterns in the field of software architecture have broader context and lack details for implementation which makes the application of architectural patterns difficult for students. To address this challenge, a pattern in the right level of abstraction with a corresponding matching technology could offer more depth and breadth based on the context of the problem beyond specific issues for example in object-oriented programming. As an example, a web development framework such as Laravel can help students learn design and implementation using MVC patterns.

In conclusion, students in SWE courses usually demonstrate some difficulties with a thorough understanding of the big picture of the system and its challenging for them to implement the systems. By using the modeling, teamwork, proper patterns and corresponding technologies, students can get hands on experience in system development while appreciating the theory of abstraction.

References

ACM. (2013). The Joint Task Force on Computing Curricula Association for Computing Machinery (ACM) IEEE Computer Society. *Computer Science Curricula 2013, Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*.

Aldaeej, A., Badreddin, O. (2016). Towards promoting design and UML modeling practices in the open source community. In: *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pp. 722–724.

Alexander, C. (1977). *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press.

Booch, G., Rumbaugh, J., Jacobson, I. (2005). *The Unified Modeling Language User Guide, 2nd edition* (July 12, 2017). Addison-Wesley.

Budgen, D. (2013). Design Patterns: Magic or Myth? *IEEE Software*.

Cetin, I., Dubinsky, E. (2017). Reflective abstraction in computational thinking. *The Journal of Mathematical Behavior*, Volume 47, 70–80. `https://doi.org/10.1016/j.jmathb.2017.06.004`

Cortez, R., Vazhenin, A. (2015). Virtual Model – View-Controller Design Pattern: Extended MVC for Service-Oriented Architecture. *IEEJ Transactions on Electrical and Electronic Engineering. IEEJ Trans 2015*, 10, 411–422. Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/tee.22101

Curry, E., Grace, P., (2008). Flexible self-management using the model-view-controller pattern. *IEEE Software*, 25(3), 84–90. DOI: 10.1109/MS.2008.60.

Dehbozorgi, N., MacNeil, S., Maher, M.L., & Dorodchi, M. (2018, October). A comparison of lecture-based and active learning design patterns in CS education. In: *2018 IEEE Frontiers in Education Conference (FIE).* IEEE, pp. 1–8.

Dehbozorgi, N. (2017, August). Active learning design patterns for CS education. In: *Proceedings of the 2017 ACM Conference on International Computing Education Research,* pp. 291–292.

Dorodchi, M., Al-Hossami, E., Nagahisarchoghaei, M., Diwadkar, RS., Benedict, A. (2019). Teaching an Undergraduate Software Engineering Course using Active Learning and Open Source Projects. In: *2019 IEEE Frontiers in Education Conference (FIE),* pp. 1–5. DOI: 10.1109/FIE43999.2019.9028517.

Dorodchi, M.M., Dehbozorgi, N., Benedict, A., Al-Hossami, E., Benedict, A. (2020). Scaffolding a team-based active learning course to engage students: A multidimensional approach. *2020 ASEE Annual Conference Content Access*. ASEE, Virtual.

Dorodchi, M., Powell, L., Dehbozorgi, N., & Benedict, A. (2020) Strategies to Incorporate Active Learning Practice in Introductory Courses. Chapter 2: Design patterns for active learning. In: Keith-Le, J. A. & Morgan, M. P. (Eds.) (2020). *Faculty experiences in active learning: A collection of strategies for implementing active learning across disciplines*. UNC Press.

Erdogmus, H., Medvidovic´, N., Paulisch, F. (2018). 50 Years of Software Engineering. *IEEE Software*, 20–24.

Gacitua, R., Sawyer, P., Gervasi, V. (2010). On the effectiveness of abstraction identification in requirements engineering. In: *18th IEEE International Requirements Engineering Conference*, pp. 5–14, DOI: 10.1109/RE.2010.12.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., & Patterns, D. (1995). *Elements of Reusable Object-Oriented Software Design Patterns*. Massachusetts: Addison-Wesley Publishing Company.

Giunchiglia, F., Walsh, T. (1992). A theory of abstraction, *Artificial Intelligence,* 57(2–3), 323–389. DOI: 10.1016/0004-3702(92)90021-O.

Kiczales, G. (1991). Towards a new model of abstraction in the engineering of software. In: *Proceedings 1991 International Workshop on Object Orientation in Operating Systems*.

Maher, ML., Dehbozorgi, N., Dorodchi, M., MacNeil, S. (2020) Chapter 10: Design patterns for active learning. In: Keith-Le, J. A. & Morgan, M. P. (Eds.) (2020). *Faculty Experiences in Active Learning: A Collection of Strategies for Implementing Active Learning across Disciplines*. UNC Press.

Medvidovic, N., Taylor, RN., Whitehead Jr., EJ. (1996). Formal modeling of software architectures at multiple levels of abstraction. *ejw*, 714, 824–2776.

Medvidovic, N., Rosenblum, DS., Redmiles, DF., Robbins, JE. (2002). Modeling software architectures in the Unified Modeling Language. *ACM Trans. Softw. Eng. Methodol.* 11(1) (January 2002), 2–57. `https://doi.org/10.1145/504087.504088`

Pop, D-P., Altar, A. (2013). Designing an MVC Model for Rapid Web Application Development. *24th DAAAM International Symposium on Intelligent Manufacturing and Automation*, 2013.

Rajput, S. (2020, December 14). What are the reasons to choose Laravel MVC for web development? *OSF preprint*. `https://doi.org/10.31219/osf.io/fgq3z`

Reenskaug, T. (2003). The Model-View-Controller (MVC) Its Past and Present.

Rugaber, S. (2006). Cataloging design abstractions. In: *Proceedings of the 2006 international workshop on Role of abstraction in software engineering (ROA '06)*. Association for Computing Machinery, New York, NY, USA, 11–18. `https://doi.org/10.1145/1137620.1137624`

Rumbaugh, J., Jacobson, I., Booch, G. (2010). *The Unified Modeling Language Reference Manual*, Addison-Wesley Professional; 2nd edition.

Salzer, H. (2010). Abstraction level hierarchy: The model and its significance for software engineering. In: *IEEE International Conference on Software Science, Technology & Engineering, SwSTE 2010,* Herzlia, Israel, June 15–16, 2010. DOI: 10.1109/SwSTE.2010.11.

Spring Web MVC, (n.d.) Retrieved from `https://docs.spring.io/spring-framework/docs/ currentreference/html/web.html#mvc on 10/30/2021`, Version 5.3.12. Last updated 2021-10-21 05:44:20 UTC.

Stumpf, RV., Teague, LC. (2006). teaching object-oriented systems analysis and design with UML. January 2006, *Information Systems Education Journal*.

Tuparov, G., Peneva, J., Asenova, P., Stanislav, I. (2007). Upskilling to object-oriented software development with UML. In: *Proceedings of the 2007 International Conference on Computer Systems and Technologies (CompSysTech '07)*. Association for Computing Machinery, New York, NY, USA, Article 70, 1–2. `https://doi.org/10.1145/1330598.1330673`

Van der Westhuizen, C., Chen, P-H., Van der Hoek, A. (2006). Emerging design: new roles and uses for abstraction. *In: Proceedings of 2006 international workshop on Role of abstraction in software engineering (ROA '06)*. Association for Computing Machinery, New York, NY, USA, 23–28. `https://doi.org/10.1145/1137620.1137626`

Wagner, F., Schmuki, R., Wagner, T., Wolstenholme, P. (2006). *Modeling Software with Finite State Machines: A Practical Approach*. Auerbach Publications; 1st edition (May 15, 2006).

Wang, Y. (2007). *Software Engineering Foundations: A Software Science Perspective*, CRC Press.

Wedyan, F., Abufakher, S. (2020). Impact of design patterns on software quality: A systematic literature review. *IET Software*, 14(1), 1–17.

M. Dorodchi is a full teaching professor of computer science at the University of North Carolina Charlotte. His research interests are in data and predictive analytics/visualization, with a focus on applications of analytics in academia and students' success. In addition, he has been extensively working on evidence-based teaching innovation, computer science education research, software engineering education, educational tool development, and K12 outreach curriculum development and broadening participation in computing. His research has been supported by a number of NSF grant project as well as State of North Carolina and local industries.

N. Dehbozorgi is an Assistant Professor of Software Engineering at Kennesaw State University, Marietta, GA. She earned her Ph.D. in Computer Science from the University of North Carolina Charlotte in 2020. She has worked in industry for several years as a software engineer, product engineer, and project manager. The scope of her research lies at the intersection of core computer science research (AI, NLP, and ML) and computing education research particularly in learning analytics and collaborative learning.

M. Fallahian is pursuing his Ph.D. in Computer Science at the University of North Carolina Charlotte. He has been working as a senior software engineer and software architect for over ten years. His research interest lies in machine learning, software engineering, big data, and database performance optimization. In addition, he has collaborated actively with researchers in designing software patterns and architecture on applied machine learning projects.

S. Pouriyeh is an Assistant Professor of Information Technology at Kennesaw State University, GA, USA. He received an M.Sc. in Information Technology Engineering from Shiraz University, and his Ph.D. in Computer Science from the University of Georgia in 2009 and 2018 respectively. His primary research interests span Federated Machine Learning, Blockchain, and Cyber Security.