Software Development for Educational Information Services Using Multilayering Semantics Adaptation

Meennapa Rukhiran, Rajamangala University of Technology Tawan-ok, Thailand
https://orcid.org/0000-0002-3880-8991
Somchai Buaroong, Rangsit University, Thailand
Paniti Netinant, Rangsit University, Thailand*
https://orcid.org/0000-0001-8376-0440

ABSTRACT

Electronic information services are robust platforms that impact daily life and facilitate new research. Increasing software modularity and reusability saves time and money. Numerous designs and programming are challenging when developing a complex educational information system. This article proposes a new paradigm of multi-dimensional information layers, aspects, functional data, and composition rules in software development to create practical educational information platforms. The proposed approach uses aspect orientation throughout educational information’s software development life cycle, from planning to implementation and evaluation. Finally, the authors demonstrate and evaluate the design model’s modularity and adaptation through fine-granularity design and class reuse. The result reduces invasive changes, promotes modularity and reuse, and eliminates duplication in component-based software design.

KEYWORDS
Adaptation, Aspect Orientation, Educational Information, Layers, Multi-Dimensional, Multilayering, Reusability, Semantics, Services, Software Development, Software Quality

INTRODUCTION

The digital transformation principle has become critical in organizing and managing university systems. The collection, processing, and dissemination of educational data are enabled by complex new data infrastructures (Williamson, 2018) that incorporate both human actors such as students, lecturers, parents, executives, and staff, as well as nonhuman actors such as departments, curriculums, and quality assurance. While software development methods have been taught at both the graduate and undergraduate levels (Wlodarski et al., 2022), the emergence of new technologies allows for the high achievement of software development challenges associated with ongoing university
and business services processes (Ibrahim et al., 2022; Macias & Aguilar-Alonso, 2022; Mandal, 2022). The quality of the software is critical when developing a complex software system. Software engineering is a term that encompasses the phases of software analysis, design, development, deployment, and maintenance. Customers’ expectations are considered throughout the software development process, from requirement analysis to maintenance. According to Mistrik et al. (2015), the scalability, flexibility, availability, modifiability, durability, and portability of customer views may incur significant implementation and maintenance costs. When a new requirement is requested to migrate from an existing application to a new launch application, software maintenance takes a long time to evolve (Martens et al., 2019). At times, software reusability is assumed to possess a range of quality attributes appropriate for specific projects (Almogahed & Omar, 2021; Guermah et al., 2021).

Based on personal financial information, a framework (Rukhiran & Netinant, 2020) is divided into three layers to support database tables, informative operations, and programming. Financial statements are classified into three categories: income, expense, and liability. The informative dimensions of multilayers have been adapted to support the challenge of multilayering semantics via informative education to develop a smart university’s educational system. During the coronavirus 2019 pandemic, a research team used Object-Oriented Design and Programming (OODP) to develop the student admission system, overcoming numerous design and programming challenges (Abohany et al., 2020). OODP is incapable of significant component reuse (Sommerville, 2014) and does not improve the internal design of a system (Malta & de Oliveira Valente, 2009). Because a single class contains numerous details and specifications, object orientation may be incompatible with effective reuse and invasive changes, such as semester modifications and payment method changes. Behaviors and classes affect interaction, inheritance, and overlap. Collaboration of reusable classes has limitations (Smiairi et al., 2022). Additionally, specific software components, such as student registrations, tuition payments, and discount promotions are repeated between classes and methods in an extending implementation. Numerous components may share the same functionality, and a class may inherit properties from other classes. In particular, numerous classes are generated to perform the same method repeatedly for various types of information management (Gulia et al., 2019). The sample issue conveys a series of coded software instructions, making them difficult for developers to comprehend, implement, adapt, and extend (Singh & Kumar, 2018). Finally, extending another educational information system to include student schedules, learning, and graduation may reveal that migrating from one complex and customized (non-reusable) software is impossible. For example, all classes are defined using the same methods, including defining the model class and inserting, updating, and retrieving data via function and method calls. It is impossible to circumvent the previous multidimensional layering implementation solutions by modifying and reusing the classes.

Apart from technical improvements, ongoing concerns exist regarding the new system’s integration. Aspect-Oriented Software Development (AOSD) is a methodology for increasing modularity and reusability in software development. Module encapsulation can aid in comprehending software applications and the resolution of trivial implementation details (Batory 2002). AOSD is one approach that can assist developers in avoiding dispersed (duplication) and tangled (dependency) code (Rukhiran & Netinant, 2020; Sanchez et al., 2010; Zhang et al., 2010). AOSD is a technique for compartmentalizing concerns that allows for the independent analysis, expression, and modularization of multiple concerns (Hovsepyan et al., 2010). The sample stages can occur at any point during the software development life cycle, from the conceptual stage to the implementation stage (Pincirolli et al., 2020; Rukhiran et al., 2021). Aspect-Oriented Programming (AOP) enables developers to partition cross-cutting concerns based on one of the single responsibility principles. Martin and Martin (2006) initially proposed the concept of a single responsibility. The fundamental concept is that each module, class, and function is responsible for a subset of functionalities and a single reason for the change, referred to as an aspect. The resulting classes and aspect abstractions do not faithfully reproduce the fundamental concept’s design (Griswold et al., 2006). The duration of the development of a program can be a representation of its singular multiplicity (Rybinski et al., 2014).
Additionally, the design of multidimensional information layering for educational information is associated with Multidimensional Separation of Concerns (MDSOC) (Batory et al., 2003; Ossher & Tarr, 2001). According to MDSOC’s brief, decomposition, and modularization can result in a variety of programs, including classes, features, aspects, and functions for supporting hyperspace designs. MDSOCs may be declared with several diverse concerns, but they must be dimensionally distinct. While AOP is concerned with the design of multidimensional spaces, the authors discovered that multidimensional layering allows for creating overlapping dimensions. The personal budget for three-dimensional space design is divided into aspect elements and functional data layering. For software execution and data retrieval, each dimension and subdimension can be used independently of the others. The results may be more efficient because multidimensional decomposition (Dong et al., 2021; Idrissi & Zerouali, 2022; Xu et al., 2021) and overlapping dimensions allow for greater flexibility (Rouached, 2021) in terms of code development (Machta et al., 2015; Wang et al., 2019), lines of code, and compiling time (Boeing et al., 2018). This article discusses the authors’ educational information operations, analysis of overlapping dimensions, and semantic definitions (Leger et al., 2015; Smiari et al., 2022). Formal verifications, such as model checking and operation semantics, are used to verify its properties and are explicit for the efficient specification and proof of systems (Jbeli & Sbai, 2021). Complex software design and development can be decomposed into multiple dimensions, with subdimensions defined for semantic relationships across various domains (Mondal et al., 2021; Opdebeeck et al., 2021; Tamine & Goeuriot, 2021).

The primary objectives of this article are to design and express semantic operations of composition rules for three-dimensional and aspectual layering of educational information services and to evaluate the development of multidimensional semantics for establishing relationships between multiple cross-cutting concerns in an educational information system. The architecture design of the educational information system is developed using a component-based approach. The composition rules are expanded in accordance with the multidimensional framework for educational information and the functional system’s aspect elements. Components can execute across multiple layers and recall methods using the multilayering semantics approach due to the information in multiple dimensions. This research investigates the adaptation of informative education design and development based on a multilayering semantics solution. The final educational information servicing system is more adaptable and flexible than the software prototypes developed using object-oriented, aspect-oriented, and multilayering approaches. The following research questions are straightforward in nature and are designed to aid in achieving the research objectives.

**Research Question One:** How can multidimensional layering be adapted to support educational information servicing systems that use semantic operations via aspect elements and functional data?

**Research Question Two:** How can semantic operations on multidimensional layering benefit the software development?

The first research question defines and implements the phases of hyperspace design, composition rules, and programming for multidimensional layering that integrates aspect elements and functional data. The second research focuses on evaluating software development and evaluation in an educational information system using a multidimensional layering of semantic operations. Thus, the multidimensional layering architecture of educational information systems can optimize for software comprehension, simplicity, reusability, and component reduction, while avoiding the development of scattered and tangled code.

**BACKGROUND**

This section summarizes the definition of the principle of single responsibility, investigates aspect-oriented approaches, and discusses multidimensional software approaches. The article focuses on
experiments with composition techniques for separating concerns, which can aid in the definition of the multidimensional layering and aspect elements used in this research contribution.

**Single Responsibility Principle**

Martin and Martin (2006) referred to the Single Responsibility Principle (SRP). According to this fundamental concept, each module, class, and function is accountable for a single aspect of functionality and a coherent reason for the change. They reason that if a class specifies multiple responsibilities, it may change for multiple reasons. As a result, accountability becomes entwined. Individuals benefit from the SRP since it assists in the task design process (Khanzadi et al., 2019). Ampatzoglou et al. (2019) have well-defined aspects of modularity. Figure 1(a) compares classes before and after applying the SRP; the lack of cohesion is unrelated to SRP use, but the coupling relation decreases. The scale and granularity of the data demonstrate the value of coupling (Schnoor & Hasselbring, 2020) and cohesion metrics for large-scale industrial software systems.

**Aspect-Oriented Approach**

To separate cross-cutting concerns, an aspect-oriented software architecture was introduced (Hoffman & Eugster, 2009). Cross-cutting concerns are functional components of a software system that span multiple components (Garcia et al., 2005; Kumara et al., 2016). In the early development process, solutions to these issues are discovered in error handling, logging, and security (Briehlau et al., 2007). There are numerous specific concerns. Aspect refers to a single concern derived from the repetition of numerous software components. A joinpoint is defined as an intersection of two dimensions that contain the aspects. A pointcut is a statement in a program that specifies how an aspect should be executed. The component requires concern separation, which is accomplished via cross-cutting points. The weaving process is the method by which concerns and fundamental codes are woven together (Lindstrom et al., 2017).

Numerous applications of the aspect-oriented approach exist, including the design of aspect-oriented learning management systems (Al-Hudhud, 2015). Katara and Katz (2007) defined an architecture model for AOSD concerns. Any aspect changes can be analyzed using the aspect views, as tracking changes across multiple aspects is challenging. The UML diagram illustrates how the perspective of an aspect changes when a method is invoked. Explicitly, any overlaps between the various cross-cutting concerns are identified. Al-Hudhud (2015) developed aspects of a management role in educational settings using AOSD principles. Due to OOP’s inability to adequately address reusability and maintainability, Aspect-Oriented Programming (AOP) is a solution technique that
enables a class to be appropriately defined within an aspect for functional decomposition (Kiczales et al., 1997). According to Hveding (2005), AOP entails three steps: identifying concerns during the design phase through decomposition of requirement specifications, implementing concerns as independent code modules, and integrating code (weaving). Numerous developers are interested in AOP principles to increase their programs’ modularity. AOP is frequently used to implement coupling (Griswold et al., 2006) and cohesion (Arpaia et al., 2010; Franca & dos Santos Soares, 2012; Hovsepyan et al., 2010). Dausend and Raschke (2016) implemented aspects of a language using abstract state machines. The formal language’s syntax and semantics are descriptive. AspectJ defines the expression for the pointcut. AspectJ is an AOP-based Java language extension that is backward compatible with the mainstream Java language. (Lung et al., 2014). During the four stages of weaving, the join points are ignored (before, around, parallel, and after). By defining the rules for a weaving process, the four stages can be assigned to the locator of the corresponding advice. The examples illustrate AOP’s potential. Additionally, when complex functionality is added during the weaving process, the original specification does not affect the interaction between the changes (Vranic et al., 2009).

Three-Dimensional Layering

To accomplish the goals of the personal financial software design (Di Pietro et al., 2021; Ma, 2021), the personal financial information is divided into three layers using the cube in Figure 2. The cube layers are composed of X, Y, and Z coordinates, with axes corresponding to expenditure, income, and liability. Each axis is further subdivided into subdimensions to accommodate individual financial data. Any coordination along an axis is designed in such a way that the cross-cutting point is explicitly stated. The layered structure makes connecting higher abstractions to more detailed specifications. An axis is layering in one dimension. Any coordinate on any axis can be called to execute by defining composition rules and semantics. For example, a user may wish to determine the total dollar amount of all expenditure records. The one dimension is then traversed recursively along the axis of the expenditure dimension. Additionally, each point on an axis can define functional data that can be used to refer to a subdimension when the user requests information about a particular subdimension. The two and three dimensions can represent the balance of financial statements, compare expenditure and income records, or compare the balances of all three types of records.

Personal financial information dimensions, subdimensions, and functional data are specified. The dimension is subdivided into several subdimensions of a relatively small size. Each subdimension has its dataset that can represent that data using related data or subdimensions. For instance, numerous

Figure 2. Three-dimensional cube belonging to personal financial information (Rukhiran & Netinant, 2020)
dimensions present a financial statement’s balance. These dimensions are combined to execute and compute a component’s process (Rukhiran & Netinant, 2020).

**Multidimensional Software Approaches**

Xiaodong et al. (1998) proposed a two-dimensional model for software development that incorporates both object-oriented and formal methods. Both object-oriented and informal methods can be used as the primary thread in the horizontal direction. The formal method can be used as an assistant thread in the vertical direction. As a result, the process enables the optimization of various software techniques.

Harrison and Ossher (1993) extended subject-oriented programming’s Multidimensional Separation of Concerns (MDSC) to the area of aspect orientation. MDSC can refer to a group of audacious goals (Ossher & Tarr, 2002). MDSC (Tarr et al., 1999) is a novel paradigm for orthogonally separating concerns in terms of compositions and decompositions. Numerous concerns intersect across multiple dimensions, including features (primary verification, persistence, and style verification), units of change, customization (adding or changing requirements in a component), data, and objects. MDSC has been proposed to bolster the hyperspace design principles (Batory et al., 2003; Khanzadi et al., 2019; Tarr et al., 1999). MDSC summarizes the decomposition and modulation into various types, including classes, features, aspects, and functions. Numerous issues have been addressed using the hyperspace approach (Benedi, 2006), resulting in the following multi-dimensions:

- A preoccupied space (language constructors such as package, interface, class, method, and attribute).
- A virtual space (as a result of multidimensional matrix identification tasks).
- An elliptical hyper slice (a functional element of specific concerns for encapsulation).
- A hyper module (comprises hyper slices and software artifacts such as components, classes, and fragments).

Functional decomposition is a versatile modularization technique (Barros, 2022) that enables users to decompose codes into any stage, such as aspects, classes, or components. The distinct concerns promote reuse, comprehension, ease of maintenance, and evolution. Enhancing traceability throughout the software development lifecycle is possible.

Constantinides et al. (2000) proposed a framework based on aspects. Concurrent systems have a complex structure that is distributed across components and aspects. The moderator of aspects is defined to evaluate the properties of any method or functionality class. As a result, the framework can avoid code transformations by simply augmenting the functional components with new attributes. Additionally, Rukhiran and Netinant (2020) proposed a system development layering strategy that is adaptive in three dimensions. The conceptual framework is intended to facilitate the layering of information in multiple dimensions. Aspect elements are multidimensional functionalities. A component is the collection of three-dimensional information layers and aspect elements. A software system’s reusability can be illustrated by subdividing its dimension into subdimensions.

**METHODOLOGY**

This section discusses the experimental methodology used to investigate the multidimensional layering of software development in informative education. In an experimental study, the Software Development Life Cycle (SDLC) is a software model that denotes the analysis phase of information concerns using formal notations, the design phase of composition rules and semantics, and the implementation phase samples. The following section discusses how the practical test and software development evaluation phases express software validation.
Concern Analysis Phase

Multidimensional Layering Transformation for Educational Information System

Three components comprise an educational information system: student, faculty, and administrator (admin). Students are divided into three categories: new students, current students, and alumni. Internal and external lectures, as well as researchers, are all considered faculty components. Officers, executives, parents, and other staff members involved in the educational information system are considered administrator components. Each component must be associated with a major. Numerous majors are organized by department. Numerous majors, including masters and doctoral degree programs, are also affiliated with a graduate school. To analyze the student admission system, general users can apply to any available programs by filling out their personal information and paying application fees. The faculty has the authority to accept or deny the student’s application. Following that, the system can automatically transfer the student to the new user registration account. The student can access the educational information system to complete tasks such as enrolling in classes, paying tuition fees, downloading and uploading documents, and informing the educational information system of a student’s graduation status. The faculties and administrators enable the monitoring of students’ activities. Informative education can incorporate X, Y, and Z coordinates to represent student, faculty, and administrator dimensions by transforming the educational information system into a multidimensional layering. Each dimension of the student, faculty, and administrator is subdivided. The composition of data sets can be used to describe the dataset for data processing, system functions, and information management in an educational information system.

Cross-Cutting Concern Analysis in Hyperspace

Hyperspace is defined as the space of multidimensional matrixes that includes a dimension of interest. Each unit or element of a software system can be subdivided into more minor concerns. Concerns can be organized using multidimensional matrixes. A concern space is made up of a collection of concerns and the aspect components that correspond to them. Hyperspace is a design term that refers to a concerned space. The hyperspace space adapts to multiple aspects to transcend other concerns. Cross-cutting points are required to carry out the dimension of concerns and associated aspects via the intersections of a single cross-cutting point. Each functional data pointcut traverses a single aspect element in a dimension. Multidimensional layering can incorporate both multidimensional education and multidimensional aspectual layering. Each aspect property is composed of aspectual elements (sub aspects) as illustrated in Figure 3(a). The separation of concerns can potentially be used as a cross-cutting component for multiple requirements. The component can access data sets via multidimensional layering. The process, dubbed weaving, integrates concern dimensions and cross-cutting concerns. When a transaction is invoked, the method call’s associated objects and conducts a transaction analysis. The numerous applications of this methodology are presented dynamically and flexibly. It is determined by the concerns and factors raised during the design process. Adaptive naming of the aspect is possible in response to method requirements.

The multi-aspect hyperspace is a representational tool for the relationship between aspects and sub-aspects. In essence, aspect properties are a collection of aspect elements. Aspects can collaborate in the following ways: an aspect property can refer to or combine with another aspect property, and an aspect element can refer to or combine with another aspect element. For instance, an aspect component composed of an aspectual semester element is used to display information about a student’s semester record. The semester aspect illustrates the relationship between the multi-aspects by combining a group of sub-aspects (the day aspect, the month aspect, and the year aspect). Each sub-aspect is defined as a member of the aspect. Each component is designed independently but in relationship to the others. The emphasis on the semester aspect is particularly evident in Figure 3(b). The educational information system has distinguished these facets. The semester aspect can occasionally be represented by the day aspect, the month aspect, or the year aspect.
Both concern dimensions and aspect elements can be thought of as having an infinite number of data points. The educational information system classifies individuals and groups of data concerns and aspect elements. As illustrated in Figure 3(a), data pertaining to student information can be expressed as an infinite series ranging from one to n. Three-dimensional layering represents a set of dimensions in a hyperspace approach. For example, one-dimensional layering may be required to obtain datasets from an axis. The x-axis is made up of datasets derived from student records and set members, where Student (S) = S1, S2, S3, ..., Sn is set. Faculty (F) = F1, F2, F3, ..., Fn to restrict the y-axis to datasets derived from faculty records. The z-axis shows datasets derived from administrator records; administrator (A) sets are denoted by the letters A1, A2, A3, ..., An. The “sets of concerns” column denotes an infinite number of planes capable of supporting three-dimensional coordinate layering. The Cartesian product is applicable to all three dataset types. Both concern dimensions and aspect elements can be defined as an infinite number of data points:

- A one-dimensional layering is a line of each axis. The formal notation sets for one layering to = \{\{S\}, \{F\}, \{A\}\}. For instance, a Cartesian product of a student is denoted by S.
- Two-dimensional layering is a coordinate plane between two axes called functional data. The formal notation sets for two layering to = \{\{S, F\}, \{F, A\}, \{S, A\}\}. For instance, a Cartesian product of a student record and a faculty record is denoted by S x F.
- Three-dimensional layering is a coordinate plane among three axes. The formal notation sets for one layering to = \{\{S, F, A\}\}. For instance, a Cartesian product of a student record, a faculty record, and an administrator record is denoted by S x F x A.

In general, quantification is accomplished through data actions on data management formulas in database tables using layering. Quantifiers are classified into two categories:

- For instance, the universal quantifier can be used to determine the total value of all student records. In Equation (1), the statement of multiple income records to “for every selection of student records in the universe, income > 1”

\[
\forall \text{student}, \text{student} > 1
\]  

(1)

**Figure 3. A hyperspace design of the dimensions of concerns crosscut the aspect elements**
For instance, the existential quantifier can determine the value of specific categories of student records. In Equation (2), the statement of multiple students records to “for some selection of student records in the universe, income > 1”

$$\exists \text{student, student} > 1$$

Thus, the concept of a Cartesian product can be extended to more than three sets. A definition is given as the concept of an ordered n-tuple. The ordered n-tuple is a set of n categories that each dimension has divided into a subdimension. The set notion uses dimension as $D = \{D_1, D_2, D_3, \ldots, D_n\}$. $Student = \{S \in D | I is a set of student records\}$. $Faculty = \{F \in D | E is a set of faculty records\}$. $Administrator = \{A \in D | L is a set of administrator records\}$. The functional data is a Cartesian product from one dimension to three dimensions.

**Composition Rules (Degree of Abstractions) of Dimensional Layering and Aspect Elements**

By decoupling the dimensions of concerns and aspect elements, an execution task during the weaving time is to combine a transaction invoked by an object. Each dimension of the information may contain one or more planes (ranging from one to three dimensions without aspects). It may incorporate dimensions and aspects (calling from one dimension to three dimensions with one aspect or more). As a result, the strategies and characteristics of compositions in Figures 3(a) and 3(b) demonstrate the use of dimensional classification. The identification task is located within a given space using a set of conversion rules. Three-dimensional layering and aspectual layering are organized according to composition rules:

**Rule One:** Each dimension’s layering is mapped to its subdimensions. Three categories have been established for the educational information system (student, faculty, and administrator concerns). A set of subdimensions is related to a set of categories. For example, the x-axis represents a dimension of student. Profiles, Curriculum, Courses, Schedules, Payments, Research, Advisor, and Graduation all represent a subdimension of the student. If the software detects any records in subdimensions, it will appropriately invoke a transaction for each component. The syntax of textual designs is expressed in Equation (3) by weaving through an object (a component). $T$ is defined as a component execution transaction composed of any quantifier type and corresponds to a single dimension of concerns. There are two types of quantifiers associated with data selection in the database table name: the universal quantifier, which selects all members ($\forall$), and the existential quantifier, which selects some members ($\exists$). In functional data, set relations are denoted by the symbol “&&”.

$$\text{ComponentName} \leftarrow \text{QuantifiersSymbol}\&\&\text{oneDimension (Name)}$$

In Equation (4), an execution example of the display component represents, which is referred to as all student records:

$$\text{Display} \leftarrow \forall \text{student}$$

**Rule Two:** Two-dimensional layering is defined as the set $D \times D = \{(D_1, D_2): (D_1 D) and (D_2 D)\}$. The two sets are arranged in two-layer pairs. For example, the y-axis represents a faculty dimension, and a set of faculty categories includes Curriculums, Profiles, Research, Courses, Payments, and Workloads. The z-axis represents an administrator dimension, and a collection of administrator categories includes Curriculums, Profiles, Payments, Workloads, and Office Hours.
A set of two-dimensional layering is returned by combining faculty and administrator records. A selections symbol employs the symbol “+” to connect two- to three-dimensional operations. Equation (5) illustrates the formula for two layering:

\[
\text{ComponentName} \leftarrow (\text{QuantifiersSymbol}\&\&\text{oneDimension(Name)}) + (\text{QuantifiersSymbol}\&\&\text{oneDimension(Name)})
\]  

(5)

In Equation (6), a sample of two-dimensional layering shows for all students and some faculty records:

\[
\text{Display} \leftarrow \forall\text{student} + \exists\text{faculty}
\]  

(6)

**Rule Three:** Three-dimensional layering is mapped onto three concerns. The Cartesian product of D, D, and D. The set D x D x D = \{(D1, D2, D3): (D1\in D) and (D2\in D) and (D3\in D)\}. The three sets are ordered, matching the layering of three dimensions. A set of layers is returned by combining student, faculty, and administrator records. The formula of three layering is shown in Equation (7):

\[
\text{ComponentName} \leftarrow (\text{QuantifiersSymbol}\&\&\text{oneDimension(Name)}) + (\text{QuantifiersSymbol}\&\&\text{oneDimension(Name)}) + (\text{QuantifiersSymbol}\&\&\text{oneDimension(Name)})
\]  

(7)

In Equation (8), a sample of three-dimensional layering shows certain student records, faculty records, and all administrator records:

\[
\text{Display} \leftarrow \exists\text{student} + \exists\text{faculty} + \forall\text{administrator}
\]  

(8)

**Rule Four:** Functional Data (FD) is an umbrella term that refers to all types of functional data. The functional data consists of a collection of intersecting three-dimensional points between Rules 1 and 3. Multilayering is accomplished by displaying multiple layers of one dimension that intersect with another dimension. For example, a display component is called to show an amount of student research records. To execute solely through the layering of student research subdimension, the weaver will invoke the functional data of national conferences, international conferences, national journals, and international journals. The syntax of a weaving combination is expressed in Equation (9).

\[
\text{ComponentName} \leftarrow \text{QuantifiersSymbol}\&\&(\text{FunctionalData(δ)} + \text{FunctionalData(γ)} + \text{FunctionalData(β)}) + \ldots
\]  

(9)

A sample of a component is given for weaving through the functional data. The amount of student research belonging to the student dimension is called the execution in Equation (10). We set δ = the national conferences, γ = the international conferences, β = the national journals, and α = the international journals:

\[
\text{Display} \leftarrow \exists\text{research(∀national conference} + \exists\text{international conference} + \forall\text{national journal} + \forall\text{international journal})
\]  

(10)

**Rule Five:** Aspect Element (AE) is a term that refers to cut across all aspect elements. The term of “multi-aspect” can refer to a concept that cut across multiple dimensions of aspects. A collection
of aspects can invoke a collaboration of aspects within a corporation. In Equation (11), the formula for cross-cutting execution can be expressed by using a symbol of “U.”

\[ \text{ComponentName} \leftarrow \text{Aspect}(\delta) \cup \text{Aspect}(\gamma) \cup \text{Aspect}(\beta) \]  

Figure 3(a) and 3(b) show the cross-cutting layer through the aspects. The date aspect assigns related to its sub aspects, such as the day aspect, the month aspect, and the year aspect. We let \( \delta = \) a day aspect, \( \gamma = \) a month aspect, \( \beta = \) a year aspect. The formula shown is the simple call of the date aspect in Equation (12), and then the execution of the sub-aspects is performed in Equation (13):

\[ \text{Display} \leftarrow \text{Aspect}(\text{date}) \]  
\[ \text{Display} \leftarrow \text{Aspect}((\text{Aspect}(\text{day}) \cup \text{Aspect(month}) \cup \text{Aspect(year)))) \]  

**Rule Six:** FD cut across AE. The function data is a set of intersecting dimensional points from Rule 1 to Rule 3. The aspect is a collection of aspectual elements, including the insert aspect, the update aspect, the delete aspect, the total aspect, and the type aspect. If \( t \) is a transaction between the function data and the aspect element, it is denoted by the notation \( t: \text{FD} \rightarrow \text{AE} \). When a transaction is invoked, execution begins. When \( t \) is viewed as an assignment from each element, \( f \) is a member of the FD that corresponds to a unique AE element. A transaction via a display component is referred to as functional data, and an aspect element or more aspects of reporting a student’s status are referred to as an aspect element. Aspects may be required to respond to a transaction by matching the target of a particular cross-cutting point at a weaver. A point of intersection is established between the functional data of one aspect via a one-to-one correspondence (1–1). The weaver may then execute from a functional to multiple aspects (1–n). \( n \) is a collection of aspects that are referred to as running. In Equation (14), the syntax of textual designs is expressed through a combination of weaving:

\[ \text{ComponentName} \leftarrow \text{QuantifiersSymbol} \& \& \text{FunctionalDataName} \\
<\langle \text{OperationOfAspect} \rangle > \text{Aspect(Name)} \]  

To illustrate one-to-one correspondence, a display component invokes a cross-cutting point for all records’ student admission status operations and a total aspect to report the total number of student admission records in Equation (15):

\[ \text{Display} \rightarrow \forall \text{student}.\text{Aspect(Total)} \]  

Additionally, a display component refers to the point of intersection for the functional data operations when displaying the record from one-to-many correspondences. In March 2021, a total aspect was required to report on all student admission records. Equation (16) assigns the formula based on the cross-cutting concerns of the functional data and numerous aspect elements:

\[ \text{Display} \rightarrow \exists \text{student}.(\text{Aspect}(\delta) \cup \text{Aspect}(\gamma) \cup \text{Aspect}(\beta)) \]  

In this sample, the weaver must call two more aspects, such as a month aspect and a year aspect, using a symbol of U. The symbols are set as \( \delta = \) the total aspect, \( \gamma = \) the month aspect, \( \beta = \) the year aspect in Equation (17):
A display component may call for more than one layering to cut across a set of aspects. For instance, the display component calls to get the advisor of a faculty for students. The names of student records and a faculty record are called to execute in the weaver. The formula is identified for displaying the concerns of the student data and many aspect elements in Equation (18):

\[
\text{Display} \rightarrow \forall \text{student}.\text{Aspect(name)} \land \forall \text{faculty}.\text{Aspect(name)} \tag{18}
\]

A probability of Cartesian products presupposes that the composition is expressed in terms of a set of dimensional layers. Seven events most likely overlap in the layering of dimensions: a set of one dimension is Student (S), Faculty (F), and Administrator (A); a set of two dimensions is S x F, S x A, and F x A; and a set of three dimensions is S x F x A. Due to the assignment of a set of aspect elements, a cross-cutting point is generated to compute the relational combination of multiple events; A cross-cutting point from one functional data to one aspect element (Cross-cutting from 1 to 1), a cross-cutting point from one functional data to two aspect elements (Cross-cutting from 1 to 2), and a cross-cutting point from one functional data to many aspect elements (Cross-cutting from 1 to n; n is several aspect elements). Table 1 illustrates the non-deterministic composition rules for one- to three-dimensional layering.

### Semantic Cross-Cutting Concerns of Implementation Phase

#### Definition of Educational Information for Programming

By dividing educational information into three categories, classes are defined for abstractions that can be used to decompose method call relationships. Figure 4 illustrates how to present a class of student information using classes. In this case, the Python programing demonstrates the article’s early design. The following student dimension examples include classes containing numerous parameters (fullname, email, status, date, and major, for example) that can be passed along via variables. Figure 5 is defined similarly to Figure 4 in terms of a class of faculty information. Due to the similar variables in each record, the authors use the separation of concerns to avoid calling the same methods multiple times. When the student class is defined, the class instantiation process automatically invokes \_\_init\_\()\) for each new instance of the student class. The class instantiation can be passed arguments, such as fullname, email, and status. The class’s instances are created to serve as a representation of records.

#### Examples of Three-Dimensional Layering Using Python Programming Abstractions

This section represents examples of three-dimensional layering using Python programming abstractions. The educational information services are implemented by three-dimensional layering to support cross-cutting concerns of layers with servicing performance. Three distinct classes are defined in Python programming and implemented via code samples. Educational data is classified into three categories (student, faculty, and administrator), each of which corresponds to a dimensional cube. Figure 6 illustrates a one-dimensional layering syntax with a single axis. The data type and name of the parameters are specified. The one-dimensional syntax template will be created using a new operator named after the class. OneDimension is the name of the class. It is possible to execute the OneDimension class first, followed by another. The class inherits from the Aspect class. The “super()” function in the OneDimension class takes two parameters (the subclass and the second parameter). As a result, the amount of code can be reduced, and the codes can be separated clearly. The declarations in the sample source code are defined independently. Using the principle of layering design, classes can be coded and implemented. The simple syntax is a style that is adaptable to any widely used programming language. To obtain two axes-related parameters, a two-dimensional layering class is defined. Figure 7 illustrates the source code for a two-dimensional layering syntax.
Table 1. Relation of layering compositions through dimensions and aspects

| Functional Data Composition | One-Dimensional Layering | Two-Dimensional Layering | Three-Dimensional Layering |
|-----------------------------|--------------------------|--------------------------|---------------------------|
| **Dimension** | S | F | A | S x F | S x A | F x A | S x F x A |
| One dimension of expenditure axis provides a set of student records = {S₁, S₂, S₃, …, Sₙ} | Reporting all amounts of student records, an operation let Display → ∃ student. Aspect (Total). | Reporting all amounts of faculty records and all amount of student records, an operation let Display → ∃ student. Aspect (Total). | Reporting all amounts of student records and faculty records and admin records, an operation let Display → ∃ student. Aspect (Total) + ∃ faculty. Aspect (Total). |
| One dimension of income axis provides a set of faculty records = {F₁, F₂, F₃, …, Fₙ} | Reporting some amounts of administrator records, an operation let Display → ∃ admin. Aspect (Total). | Reporting some amounts of student records and some amount of admin records, an operation let Display → ∃ student. Aspect (Total) + ∃ admin. Aspect (Total). | Reporting some amount of faculty records and some amount of admin records, an operation let Display → ∃ faculty. Aspect (Total) + ∃ admin. Aspect (Total). |
| Two dimensions product student and faculty records. S x F = {(S₁, F₁), (S₂, F₁), (S₃, F₁), …, (Sₙ, F₁)} | Reporting an amount of faculty records in March, an operation let Display → ∃ faculty. Aspect (Total) U Aspect (Month)). | Comparison an amount of student and faculty records in March, an operation let Display → ∃ student. Aspect (Total) U Aspect (Month)) + ∃ faculty. Aspect (Total) U Aspect (Month)). | Comparison an amount of student and faculty records in March, an operation let Display → ∃ student. Aspect (Total) U Aspect (Month) + ∃ faculty. Aspect (Total) U Aspect (Month)). |
| Two dimensions product student and faculty records and admin records. S x A = {(S₁, A₁), (S₂, A₁), (S₃, A₁), …, (Sₙ, A₁)} | Reporting an amount of student and liability records in March, an operation let Display → ∃ student. Aspect (Total) U Aspect (Month)). | Comparison an amount of student and admin records in March, an operation let Display → ∃ student. Aspect (Total) U Aspect (Month) + ∃ admin. Aspect (Total) U Aspect (Month)). | Comparison an amount of student, faculty and admin records in March, an operation let Display → ∃ student. Aspect (Total) U Aspect (Month)). |

continued on following page
Table 1. Continued

| Functional Data Composition | One-Dimensional Layering | Two-Dimensional Layering | Three-Dimensional Layering |
|-----------------------------|---------------------------|--------------------------|----------------------------|
|                             | S | F | A | S x F | S x A | F x A | S x F x A |
| Cross-cutting from 1 – n    |   |   |   |   |   |   |                      |
| (A functional data to a number of aspects) |   |   |   |   |   |   |                      |
| Reporting an amount of admin records at a particular time, an operation let Display \( \rightarrow \exists \) student. (Aspect (Total) U Aspect (Month) U...) |   |   |   |   |   |   |                      |
| Reporting an amount of admin records at a particular time, an operation let Display \( \rightarrow \exists \) faculty. (Aspect (Total) U Aspect (Month) U...) |   |   |   |   |   |   |                      |
| Reporting an amount of admin records at a particular time, an operation let Display \( \rightarrow \exists \) admin. (Aspect (Total) U Aspect (Month) U...) |   |   |   |   |   |   |                      |
| Cross-cutting from 1 – n    |   |   |   |   |   |   |                      |
| (A functional data to a number of aspects) |   |   |   |   |   |   |                      |
| Reporting an amount of admin records at a particular time, an operation let Display \( \rightarrow \exists \) student. (Aspect (Total) U Aspect (Month) U...) |   |   |   |   |   |   |                      |
| Reporting an amount of admin records at a particular time, an operation let Display \( \rightarrow \exists \) faculty. (Aspect (Total) U Aspect (Month) U...) |   |   |   |   |   |   |                      |
| Reporting an amount of admin records at a particular time, an operation let Display \( \rightarrow \exists \) admin. (Aspect (Total) U Aspect (Month) U...) |   |   |   |   |   |   |                      |

Figure 4. A class skeleton of student information

```python
class Student:
    def __init__(self, full_name, email, status, application_date, major, education, ...):
        self.fullname = full_name
        self.email = email
        self.status = status
        self.application_date = application_date
        self.major = major
        ...

student_1 = Student(‘John William’, ‘john_wi@gmail.com’, ‘completed’, ‘02-03-2021’, ‘IT’, ...)
student_2 = Student(‘Adam Scott’, ‘adam_sc@gmail.com’, ‘completed’, ‘13-03-2021’, ‘IT’, ...)
student_3 = Student(‘Jane Park’, ‘jan_pa@gmail.com’, ‘checking’, ‘13-04-2021’, ‘IT’, ...)
```

Figure 5. A class skeleton of faculty information

```python
class Faculty:
    def __init__(self, full_name, email, status, department, major, education, ...):
        self.fullname = full_name
        self.email = email
        self.status = status
        self.department = department
        self.major = major
        ...

faculty_1 = Faculty(‘Nicely Wong’, ‘nicely_ws@gmail.com’, ‘on location’, ‘Civil Engineering’, ‘CE’, ...)
faculty_2 = Faculty(‘Malee Wiley’, ‘malee_wi@gmail.com’, ‘working’, ‘Accounting’, ‘ACC’, ...)
faculty_3 = Faculty(‘Michael Skyway’, ‘michael_sk@gmail.com’, ‘working’, ‘Computing’, ‘IT’, ...)
```
Three parameters are defined for a class that implements three-dimensional layering. Figure 8 defines a three-dimensional layering syntax.

**Multidimensional Software Approaches**

Numerous aspects can be found in code programs that ensure they are clean and easy to reuse. The declarations’ primary purposes determine it. This software design is geared toward developing educational information software, and the educational content is organized in a three-dimensional layering structure. Extending this work through separation of concerns poses a significant challenge in optimizing single responsibilities and avoiding function and method calls, method execution, and data recording, being all concerned with the same thing. As a result, educational and functional data are excluded from the definition of aspectual properties. The term “aspect” refers to the aspect element. Aspects are defined as a collection of intersecting properties related to a system’s functionalities or methods. Throughout the software system, the characteristics are specified repeatedly. They are necessary due to other considerations and multiple calls within the same functional design. To determine whether or not to store student, faculty, and administrator data in databases, similar information such as the full name, the email address, the status, the application date, and the education can be frequently scoped in any record type containing educational information. As a result, the core functionality is based on a crude operation that manages transactions (data creation, retrieval, update, and deletion). By defining the functionality for each aspect, various aspects such as date, day, month, and year, full name, email, status, application date, and education, as well as insertion, deletion,
updating, and connection, are established. The examples in Figures 9 and 10 illustrate sample source code for aspect declarations.

**Implementation of Composition Rules**

The statement of weaving defines how educational information classes and program components are interwoven. The educational data categories, such as student and faculty, are defined. The fundamental concepts required to support three-dimensional layering and aspect elements composition rules are compiled. The sample combination of Equation (16) is illustrated in Figure 11. The statement in the display component represents the process of weaving in order to display the student’s grade. The OneDimension class represents the functional data. The grade class is specified as the aspect element. The “getgrade” function is used to define the aspect property’s sample size. Finally, the combined classes are woven into the display component. Figure 12 illustrates the proof of Equation (18) in its entirety. The display component displays information about students and faculty in the department and major. The statement of the display component is represented by weaving through multiple layers. To cut across a collection of facets, two-dimensional layering is used. The aspect property (the “getadvisor” function) is defined to facilitate this execution.

**RESULTS AND DISCUSSION**

**Practical Validation Using Key Qualities**

The authors’ artifacts illustrate novel paradigms for coding modeling, design, and implementation. The degree of abstraction is defined through three-dimensional layering, thereby supporting the composition rules. Essential classes are modularized individually in the preceding section. Each class contains a distinct set of methods. Specifically designed are the educational classes (student, faculty, and administrator), the multidimensional class (functional data), and the aspect elements and properties (aspects). The class definitions do not overlap due to cross-cutting concerns based on dimension layering, the composition of functional data and aspect weaving. References to other

---

```python
class Education:
    def getdeparment(fullname):
        return 'Your department is {}'.format(department)
    def getgrade(date, fullname, department):
        return 'Your grade is {}'.format(grade)

Figure 9. An implement of students' department selection in IT department as a skeleton class
```

```python
class Date:
    def getdate(date1, date2, semester, credit):
        # select semester, course, credit from database between (date1 and date2)
        print ('Your course from {} to {}'.format(date1, date2))
        for i in range(date1, date2):
            print ('Course name {} credit {} grade {}'.format(course, credit, grade))

Figure 10. A date aspect implemented as a skeleton class
```
Functional data classes may be contained within a functional data class. Furthermore, the aspect class may conflict with other aspect classes. The context evaluation is concerned with achieving software test coverage. As a result, developers can more easily design modules for compilation and select classes for execution during the running process.

---

### Figure 11. A definition of a class skeleton and the subsequent execution of one-dimensional syntax

| Student Information Class: | One Dimension (Functional Data) | Total (Aspect Property: Getgrade Elements) | Let Display → VStudent.Aspect(Grade) (Weaving) |
|----------------------------|---------------------------------|-------------------------------------------|-----------------------------------------------|
| classStudent:              | def __init__(self, fullname, email, status, application_date, major, education, ...):  
|                           | self.name = name  
|                           | self.fullname = fullname  
|                           | self.email = email  | classOneDimension (Aspect):  
|                           | def __init__(self):  
|                           | super(OneDimension, self).__init__()  | classGrade:  
|                           | def getgrade(date, fullname, department):  
|                           | return ‘Your grade is {}’.format(grade)  | class  
|                           | Display(name):  
|                           | print(OneDimension (getgrade(date, student.fullname, student.department)))  |

**Method Call:**

```
>>Display("Getgrade")
```

**Result:**

```
Your grade is 3.42
```

---

### Figure 12. A definition of a class skeleton and the subsequent execution of two-dimensional syntax

| Student Information Class: | Two Dimensions (Functional Data) | Total (Aspect Property: Getadvisor Elements) | Let Display → VStudent.Aspect(Advisor) & VFaculty.Aspect(Advisor) |
|----------------------------|---------------------------------|-------------------------------------------|-----------------------------------------------|
| class Student:             | def __init__(self, fullname, email, status, application_date, major, education, ...):  
|                           | self.fullname = fullname  
|                           | self.email = email  | classTwoDimension (Aspect):  
|                           | def __init__(self):  
|                           | super(TwoDimension, self).__init__()  | classAdvisor:  
|                           | def getadvisor(advisorname, studentname):  
|                           | return '{}advisor is {}\'.format(studen
tname, faculynname)  | class  
|                           | Display(name):  
|                           | print(TwoDimension (getadvisor(faculty.fullname, student.fullname)))  |
| classFaculty:              | def __init__(self, fullname, email, status, department, major, education, ...):  
|                           | self.fullname = fullname  
|                           | self.email = email  |

**Method Call:**

```
>>Display("Getadvisor")
```

**Result:**

```
John Williamadvisor is Michael Skyway
```
**Improving Comprehensibility**

A class is an afterthought by fragmenting designs and codes for individual methods. Layering dimensions, functional data, and aspects are all explicitly declared classes. Three-dimensional classes are used to classify dimensions. Functional data is structured so that it can be accessed via subdimensions. Independent of aspect elements, base codes are declared. The aspect elements of each unique function are composed of the aspect properties (sub-aspects).

**Reducing Complexity**

A decrease in complexity entails a reduction in relational class. The concept appears to be feasible in keeping the code clean and simple to understand when it comes to layer execution. By decomposing the complexity of the software design and programming phases consistently into methods and functions, the complexity of the software design and programming phases can be decomposed. Numerous abstraction layers demonstrate the granularity with which functional data and aspects are designed. Thus, programming methods can be implemented using the weaving expression and then dynamically changed within a composite application.

**Avoiding Scattered and Tangled Code**

Separation of concerns is a design and implementation technique. As illustrated in Figure 13, cross-cutting concerns are characteristics of software programs that help avoid scattering (code duplication) and tangling (significant dependency). By allowing for the provision of concerns in a single aspect, aspect-oriented programming enables one to achieve a higher level of modularization. Multiple modules are not used to contain the scattering. Multiple concerns can share the aspect property (Getgrade). Additionally, the term “tangling” can refer to a single module’s treatment of the same issue. By omitting this step, a fine granularity enables the compilation of a single method to traverse another class. As a result, other components may not contain identical code.

**Increasing Reusability**

As illustrated in Figure 14, refactoring can increase the reusability of classes. Simple aspect element refactoring (AlOmar et al., 2021) enables component-level calls to share a typical class. The total aspect (method), in this case, is the “TotalStudent” function. While the OneDimension class can use the “TotalFaculty” function to display the total number of faculty records, other classes can be used to display the total number of student records. Additionally, aspect classes can be reused by other functional classes. Three-dimensional classes and educational class principles, according to this article, enable one to maximize software reuse and migration to newly developed services.

![Figure 13. Fine granularity of layering artifact](image-url)
While the multi-dimensional layering decomposes the functions and similar concerns, this proposed article on aspect-orientated programming is distinct from those used in various other types of research. Numerous previous works (Hoffman & Eugster, 2009, 2013; Vranic et al., 2009) have focused on developing a set of syntactic logic for implementing concerns located before, after, and around join points. The principle of single accountability has been applied (Khanzadi et al., 2019; Rana & Khonica, 2021). The repeated explicit classes define methods and functionality not associated with the software system.

**Evaluation of Development Results**

On a web-based platform, the educational software was developed in C#.Net. There are numerous features included in the educational information system. Each functionality is dependent on a variety of components to operate properly. When using the Rapid Application Development (RAD) methodology to develop a prototype for users to test and verify functionality, each prototype must adhere to a common aesthetic design and usage. The following component ensures that users are delighted and at ease when interacting with the web-based application, and that the system does not interfere with the new normal processes. If users experience discomfort and dissatisfaction while interacting with the system, the authors must reconsider, redesign, implement, evaluate, and represent the system to users for feedback. The educational information system is composed of 16 primary components. As shown in Table 2, each component is subdivided into subcomponents. The initial prototype, which was created and developed object-oriented, contained 191 subcomponents. Each modification to the prototype will necessitate the replacement of numerous components that share common functionality and system properties. When the functionality of a component is altered, or a system property is omitted from other components, the component’s data becomes inconsistent, which may result in system errors. The successor-programming prototype has no missing information flows, processes, user interfaces, or databases, designed and developed using an aspect-oriented approach and the Information Flow Diagram (IFD). On the other hand, the authors have lost track of component interactions, trading data, redundant functionality, and system properties, making changes difficult. As a result, the second prototype underwent numerous revisions before being made available to users.

Prototypes for improvement were designed and developed using an aspect-oriented approach and the concept of multilayering semantics. This article possesses the experience necessary to analyze, reconsider, redesign, implement, and correct, resulting in decreased development time as a result of improved comprehension, less tightly coupled software, and increased cohesion. Each layer of the multilayer semantics categorizes components as either aspectual or functional data components. Each component clearly and ultimately defines its inputs, outputs, layers, operations, and constraints. This technique enables developers and designers to effectively analyze and refactor implausible
components within each layer. The multilayering semantics revealed that when combined with an aspect-oriented approach, the third prototype enables the application to be designed and developed with fewer components, interactions, data, and processes while retaining the same functionality and properties. The number of identical components in Table 2 can be decreased from 191 to 113. Thus, the final prototype of informative, educational software based on a web-based smartphone device contains one-third of the components found in the third prototype, a 38.76 percent reduction in the component count.

Figure 15 compares the number of educational information software components based on a web-based application designed and developed using a variety of successor software development approaches to the most recent design and development prototype that utilized the component-based multilayering semantics approach. The article proposes educational information software components using multilayering semantics and an aspect-oriented approach. Although multilayering semantics are rarely used to define a component’s property within a layer, the component-based system can be simplified, which is advantageous when designing and developing complex software. The advantage of component characteristics is that they facilitate the separation of aspects and functional data during the weaving execution of each layer. Components included datasets, functional data, aspect elements, and configurations accessible via various layers. The various concerns and aspects can easily represent common component layering relationships. Figure 15 depicts a selection of reducing components that can display significantly different information. However, software development using multilayering semantics demonstrates the creation of components at each layer’s crosscutting point. A component acting as an object is required to weave at joining points. As a result, a syntax

| Components              | OOA APPROACH | OOA + IFD APPROACH | AOA + Multilayering Semantics | Component Reduction From OOA APPROACH | %Reduction |
|-------------------------|--------------|--------------------|--------------------------------|----------------------------------------|------------|
| Welcome Screens         | 15           | 10                 | 10                             | 5                                      | 33.33%     |
| User Registering       | 7            | 5                  | 4                              | 3                                      | 42.86%     |
| User Authentication    | 6            | 4                  | 3                              | 3                                      | 50.00%     |
| Multilanguage          | 12           | 9                  | 9                              | 3                                      | 25.00%     |
| Student Information    | 14           | 11                 | 9                              | 5                                      | 35.71%     |
| Faculty Information    | 14           | 11                 | 9                              | 5                                      | 35.71%     |
| Administrator Information | 14         | 11                 | 9                              | 5                                      | 35.71%     |
| Menu Screens           | 6            | 4                  | 4                              | 2                                      | 33.33%     |
| Enrollment             | 19           | 17                 | 14                             | 5                                      | 26.32%     |
| Schedule               | 6            | 6                  | 3                              | 3                                      | 50.00%     |
| Account Management     | 10           | 9                  | 5                              | 5                                      | 50.00%     |
| Department             | 14           | 12                 | 8                              | 6                                      | 42.86%     |
| School                 | 14           | 12                 | 8                              | 6                                      | 42.86%     |
| Program                | 14           | 12                 | 8                              | 6                                      | 42.86%     |
| Tuition Calculation    | 18           | 16                 | 7                              | 11                                     | 61.11%     |
| User Profile           | 8            | 7                  | 7                              | 1                                      | 12.50%     |
| **Total**              | **191**      | **156**            | **113**                        | **50**                                 | **38.76%** |
overview of explicit joint points should be defined to aid in the translation of the multilayering semantics design to programming.

This article uses the IEEE standard 1016-2009 (2009) for information technology to describe software design descriptions (SDDs). IEEE Std 1016-2009 specifies that an SDD should be organized into a few design views. Each perspective addresses a distinct set of design concerns raised by stakeholders. Thus, to provide a more comprehensive view of a design, design entities such as components, classes, data stores, and processes can be used to capture all critical elements required for supporting design views. Additionally, the composition is refined into new perspectives, as stated in the summary of design perspectives. Physically, this article can be expressed through physical designs; logically, it can be expressed through functional decompositions. Additionally, the design of information partitioning should be improved.

The multilayering semantics technique has demonstrated that the component-based model focuses on fewer decompositions of tangling components while preserving perfectly consistent previous system operations. Multilayer semantics enables component interoperations within individual functional or aspectual components while accurately representing critical communications and semantics. Additional concerns about component-based software development should be studied, such as system performance, reusability, extensibility, and adaptability. The quality concerns must be addressed before determining how this technique may accomplish the final results.

**CONCLUSION**

Multidimensional layering is an optimal software design and development approach that explicitly takes classes, methods, and components into account and significantly improves software quality, especially in design and development. The multidimensional layering objective can be supplemented and adapted by rules for semantic integration of educational information software systems into the

---

Figure 15. Comparison of Components in the web-based application developed by different approaches
conceptual framework design. Composition rules established a framework for both multidimensions and aspects based on concern separation. The hyperspace approach is defined in this study as a technique for combining layer designs. Semantic operations can express the overlapping dimensions of three-dimensional and aspectual layering. A set of conversion rules is required for multidisciplinary strategies in compositions, such as a single aspect cutting across all other aspects, a single functional data cutting across all other functional data, a single functional data cutting across all other aspects, and a multi-cross-cutting of dimensions, functional data, and aspects. The following evaluations demonstrate the importance of software quality improvement during development.

Multiple cross-cutting concerns have explicit semantics that can be computed using in-class components—conversion of the semantic cross-cutting concerns to a Python programming executable. The Python program is a simple programming language used to define the components. There is extensive discussion of the educational, three-dimensional, and aspect element classes. The woven code is constructed using dimension layering to ensure that the appropriate classes are selected. The codes are capable of running at a high rate of speed. The source code is provided and evaluated using a cleanly modularized syntax to evaluate the aspect-oriented programming method and functionality. Reusability of classes exemplifies how dynamically woven aspect classes can be. Multiple educational classes can be accessed through the same aspect class. Evaluating the multidimensional layering of educational information systems makes it possible to eliminate concerns and components dispersed across modules and entangled in the final software prototypes. Future research will concentrate on implementing concurrency in software components throughout the educational information systems development process. Numerous component architectures in the educational information software system should be analyzed, designed, and implemented. Additionally, further research can be conducted on developing a meta-aspect model to support the metadata associated with multidimensional layering.
REFERENCES

Abohany, A. A., Rizk-Allah, R. M., Mosa, D. T., & Hassanien, A. E. (2020). A novel approach for solving a fully rough multi-level quadratic programming problem and its application. *International Journal of Service Science, Management, Engineering, and Technology, 11*(4), 137–165. doi:10.4018/IJSSMET.2020100109

Al-Hudhud, G. (2015). Aspect oriented design for team learning management system. *Computers in Human Behavior, 51*, 627–631. doi:10.1016/j.chb.2015.01.032

Almogahed, A., & Omar, M. (2021). Refactoring techniques for improving software quality: Practitioners’ perspectives. *Journal of Information and Communication Technology, 20*(4), 511–539. doi:10.32890/jict2021.20.4.3

AlOmar, E. A., Mkaouer, M. W., Newman, C., & Ouni, A. (2021). On preserving the behavior in software refactoring: A systematic mapping study. *Information and Software Technology, 140*, 106675. doi:10.1016/j.infsof.2021.106675

Ampatzoglou, A., Tsintzira, A. A., Arvanitou, E. M., Chatzigeorgiou, A., Stamelos, I., Moga, A., Heb, R., Matei, O., Tsiridis, N., & Kehagias, D. (2019). Applying the single responsibility principle in industry: Modularity benefits and trade-offs. In Evaluation and Assessment on Software Engineering (pp. 347-352). doi:10.1145/3319008.3320125

Arpaia, P., Bernardi, M. L., Di Lucca, G., Inglese, V., & Spiezia, G. (2010). An aspect-oriented programming-based approach to software development for fault detection in measurement systems. *Computer Standards & Interfaces, 32*(3), 141-152. doi:10.1016/j.csi.2009.11.009

Barros, O. (2022). A management and enterprise architecture framework for comprehensive structure design of complex services. *International Journal of Service Science, Management, Engineering, and Technology, 13*(1), 1–24. doi:10.4018/IJSSMET.290335

Batry, D. (2002). Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodology, 11*(2), 215–255. doi:10.1145/505145.505148

Batory, D., Liu, J., & Sarvela, J. N. (2003, September). Refinements and multidimensional separation of concerns. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (pp. 48-57). doi:10.1145/940071.940079

Benedi, J. P. (2006). *PRISMA: Aspect-oriented software architectures* (Doctoral dissertation). Polytechnic University of Valencia.

Boeing, P., Leon, M., Nesbeth, D. N., Finkelstein, A., & Barnes, C. P. (2018). Towards an aspect-oriented design and modelling framework for synthetic biology. *Processes, 6*(9), 167. 10.3390/pr6090167

Brichau, J., Chitchyan, R., Rashid, A., & D’Hondt, T. (2007). Aspect-oriented software development: An introduction. Wiley Encyclopedia of Computer Science and Engineering. doi:10.1002/9780470050118.ecse022

Constantinides, C. A., Bader, A., Elrad, T. H., Netinant, P., & Fayad, M. E. (2000). Designing an aspect-oriented framework in an object-oriented environment. *ACM Computing Surveys (CSUR), 32*(1es). 10.1145/351936.351978

Dausend, M., & Raschke, A. (2016). Towards a comprehensive extension of abstract state machines for aspect-oriented specification. *Science of Computer Programming, 131*, 22–41. doi:10.1016/j.scico.2016.04.007

Di Pietro, R., Raponi, S., Caprolu, M., & Cresci, S. (2021). New dimensions of information warfare. In *New Dimensions of Information Warfare* (pp. 1–4). Springer. doi:10.1007/978-3-030-60618-3_1

Dong, X., Cai, L., Fan, H., & Wang, X. (2021, November). Multi-dimensional analysis of university library borrowing based on big data and statistical analysis. In *International Conference on Machine Learning and Big Data Analytics for IoT Security and Privacy* (pp. 756-762). Springer. doi:10.1007/978-3-030-89511-2_100

dos Santos Soares, M. (2012). A systematic review on evaluation of aspect-oriented programming using software metrics. *ICEIS, (2)*, 77-23. 10.5220/0003985300770083
Garcia, A., Kulesza, U., Sant’Anna, C., Chavez, C., & de Lucena, C. J. (2005, July). Aspects in agent-oriented software engineering: Lessons learned. In International Workshop on Agent-Oriented Software Engineering (pp. 231-247). Springer. doi:10.1007/11752660_18

Griswold, W. G., Sullivan, K., Song, Y., Shonle, M., Tewari, N., Cai, Y., & Rajan, H. (2006). Modular software design with crosscutting interfaces. IEEE Software, 23(1), 51–60. doi:10.1109/MS.2006.24

Guermah, H., Guermah, B., Fissaa, T., Hafidi, H., & Nassar, M. (2021). Dealing with context awareness for service-oriented systems: An ontology-based approach. International Journal of Service Science, Management, Engineering, and Technology, 12(4), 110–131. doi:10.4018/IJSSMET.2021070107

Gulia, P., Khari, M., & Patel, S. (2019). Metrics analysis in object-oriented and aspect-oriented programming. Recent Patents on Engineering, 13(2), 117–122. doi:10.2174/1872212112666180831115458

Guermah, H., Guermah, B., Fissaa, T., Hafidi, H., & Nassar, M. (2021). Dealing with context awareness for service-oriented systems: An ontology-based approach. International Journal of Service Science, Management, Engineering, and Technology, 12(4), 110–131. doi:10.4018/IJSSMET.2021070107

Harrison, W., & Ossher, H. (1993, October). Subject-oriented programming: A critique of pure objects. In Proceedings of the 8th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (pp. 411-428). doi:10.1145/165854.165932

Hoffman, K., & Eugster, P. (2009). Cooperative aspect-oriented programming. Science of Computer Programming, 74(5-6), 333–354. doi:10.1016/j.scico.2009.01.003

Hoffman, K., & Eugster, P. (2013). Trading obliviousness for modularity with cooperative aspect-oriented programming. ACM Transactions on Software Engineering and Methodology, 22(3), 1–46. doi:10.1145/2491509.2491516

Hovsepyan, A., Scandariato, R., Baelen, V., Berbers, Y., & Joosen, W. (2010). From aspect-oriented models to aspect-oriented code? The maintenance perspective. In 9th International Conference on Aspect-Oriented Software Development (pp. 85-96). doi:10.4018/IJSSMET.2021070105

Hxeding, J. C. (2005). An aspect-oriented approach to adaptive systems (Master’s thesis). Institutt for datateknikk og informasjonsvitenskap.

Ibrahim, M., Yusheng, K., & Adam, D. R. (2022). Linking service innovation to organisational performance: Mediating role of employee productivity and job satisfaction. International Journal of Service Science, Management, Engineering, and Technology, 13(1), 1–16. doi:10.4018/IJSSMET.295558

Idrissi, K., & Zerouali, E. H. (2022). Excursiveness approach to multi-dimensional moment problems. Annals of Functional Analysis, 13(1), 1–19. doi:10.1007/s43034-021-00149-2

IEEE Standard for Information Technology--Systems Design--Software Design Descriptions. (2009). IEEE STD 1016-2009. .10.1109/IEEESTD.2009.5167255

Jbeli, N., & Sbai, Z. (2021). On improving model checking of time petri nets and its application to the formal verification. International Journal of Service Science, Management, Engineering, and Technology, 12(4), 68–84. doi:10.4018/IJSSMET.2021070105

Katara, M., & Katz, S. (2007). A concern architecture view for aspect-oriented software design. Software & Systems Modeling, 6(3), 247–265. doi:10.1007/s10270-006-0032-x

Khanzadi, M., Shahbazi, M. M., Arashpour, M., & Ghosh, S. (2019). The less agents, the more schedule reliability: Examination of single-point responsibility model in design management. International Journal of Civil Engineering, 17(8), 1307–1316. doi:10.1007/s40999-018-00389-9

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. M., & Irwin, J. (1997, June). Aspect-oriented programming. In European Conference on Object-Oriented Programming (pp. 220-242). Springer. doi:10.1007/BFb0053381

Kumar, A., Kumarb, A., & Iyyappanna, M. (2016). Applying separation of concern for developing software using aspect oriented programming concepts. Procedia Computer Science, 85, 906–914. doi:10.1016/j.procs.2016.05.281

Leger, P., Tanter, E., & Fukuda, H. (2015). An expressive stateful aspect language. Science of Computer Programming, 102, 108–141. doi:10.1016/j.scico.2015.02.001
Lindstrom, B., Offutt, J., Sundmark, D., Andler, S. F., & Pettersson, P. (2017). Using mutation to design tests for aspect-oriented models. *Information and Software Technology, 81*, 112-130. doi:10.1016/j.infsof.2016

Lung, C. H., Ajila, S. A., & Liu, W. L. (2014). Measuring the performance of aspect oriented software: A case study of leader/followers and half-sync/half-async architectures. *Information Systems Frontiers, 16*(5), 853–866. doi:10.1007/s10796-013-9423-z

Ma, R. (2021, May). Design and implementation of financial information system for mobile devices. *Journal of Physics: Conference Series, 1915*(4), 042010. doi:10.1088/1742-6596/1915/4/042010

Machta, N., Bennani, M. T., & Benahmed, S. (2015). NOLE: An AOM weaver for aspect oriented modeling of real-time system. *Procedia Computer Science, 46*, 742–748. doi:10.1016/j.procs.2015.02.141

Macias, C. M., & Aguilar-Alonso, I. (2022). Assessment of contributions of the methodology for the construction of a catalog of information technology services aimed at public entities. *International Journal of Service Science, Management, Engineering, and Technology, 13*(1), 1–28. doi:10.4018/IJSSMET.287911

Malta, M. N., & de Oliveira Valente, M. T. (2009). Object-oriented transformations for extracting aspects. *Information and Software Technology, 51*(1), 138-149. doi:10.1016/j.infsof.2008.02.001

Mandal, P. C. (2022). Service excellence in marketing for firms: Strategies and initiatives. *International Journal of Service Science, Management, Engineering, and Technology, 13*(1), 1–22. doi:10.4018/IJSSMET.290331

Martens, A., Book, M., & Gruhn, V. (2019). A data decomposition method for stepwise migration of complex legacy data. *Software, Practice & Experience, 49*(2), 214–232. doi:10.1002/spe.2647

Martin, R. C., & Martin, M. (2006). *Agile principles, patterns, and practices in C#* AGILE PRIN PATTS PRACTS C#. Pearson Education.

Mistrik, I., Soley, R. M., Ali, N., Grundy, J., & Tekinerdogan, B. (Eds.). (2015). *Software quality assurance: In large scale and complex software-intensive systems*. Morgan Kaufmann.

Mondal, A. K., Roy, C. K., Schneider, K. A., Roy, B., & Nath, S. S. (2021). Semantic slicing of architectural change commits: Towards semantic design review. In *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (pp. 1-6). doi:10.1145/3475716.3484487

Opdebeeck, R., Zerouali, A., Velázquez-Rodríguez, C., & De Roover, C. (2021). On the practice of semantic versioning for ansible galaxy roles: An empirical study and a change classification model. *Journal of Systems and Software, 182*, 111059. doi:10.1016/j.jss.2021.111059

Ossher, H., & Tarr, P. (2001). Using multi-dimensional separation of concerns to (re) shape evolving software. *Communications of the ACM, 44*(10), 43–50. doi:10.1145/383845.383856

Ossher, H., & Tarr, P. (2002). Multi-dimensional separation of concerns and the hyperspace approach. In *Software Architectures and Component Technology* (pp. 293-323). doi:10.1007/978-1-4615-0883-0_10

Pinciroli, F., Justo, J. L. B., & Forradellas, R. (2020). Systematic mapping study: On the coverage of aspect-oriented methodologies for the early phases of the software development life cycle. *Journal of King Saud University–Computer and Information Sciences, 1*, 1-14. 10.1016/j.jksuci.2020.10.029

Rana, M. E., & Khonica, E. (2021). Impact of design principles and patterns on software flexibility: An experimental evaluation using Flexible Point (FXP). *Journal of Computational Science, 17*(7), 624–638. doi:10.3844/jcssp.2021.624.638

Rouached, M. (2021). Web service composition security: A three-dimensions overview. *International Journal of Service Science, Management, Engineering, and Technology, 12*(3), 154–174. doi:10.4018/IJSSMET.2021050109

Rukhiran, M., & Netinant, P. (2020). A practical model from multidimensional layering personal financial information framework to using mobile software interface operations. *Journal of Information and Communication Technology, 19*(3), 321–349. doi:10.32890/jict2020.19.3.2

Rukhiran, M., Netinant, P., & Eldad, T. (2021). Multiconcerns circuit component diagram apply to improve on software development: Empirical study of house bookkeeping mobile software. *Journal of Current Science and Technology, 11*(2), 240–260. doi:10.14456/jcst.2021.25
Rybinski, K., Jarzebowski, N., Smialek, M., Nowakowski, W., Skrzypek, L., & Labecki, P. (2014). Generating graphical user interfaces from precise domain specifications. *e-Informatica. Software Engineering Journal, 8*(1), 39–52. doi:10.5277/e-Inf140103

Sanchez, P., Moreira, A., Fuentes, L., Araujo, J., & Magno, J. (2010). Model-driven development for early aspects. *Information and Software Technology, 52*(3), 249-273. infsof.2009.09.00110.1016/j

Schnoor, H., & Hasselbring, W. (2020). Comparing static and dynamic weighted software coupling metrics. *Computers, 9*(2), 24. doi:10.3390/computers9020024

Singh, R., & Kumar, A. (2018). Identifying various code-smells and refactoring opportunities in object-oriented software system: A systematic literature review. *International Journal on Future Revolution in Computer Science & Communication Engineering, 8*(March), 62–74. doi:10.1016/j.infsof.2014.08.002

Smiari, P., Bibi, S., Ampatzoglou, A., & Arvanitou, E. M. (2022). Refactoring embedded software: A study in healthcare domain. *Information and Software Technology, 143*, 106760. doi:10.1016/j.infsof.2021.106760

Sommerville, I. (2014). *Software engineering* (10th ed.). Pearson Education, Inc.

Tamine, L., & Goeuriot, L. (2021). Semantic information retrieval on medical texts: Research challenges, survey, and open issues. *ACM Computing Surveys, 54*(7), 1–38. doi:10.1145/3462476

Tarr, P., Ossher, H., Harrison, W., & Sutton, M. (1999). N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No. 99CB37002)* (pp. 107-119). IEEE doi:10.1109/302405.302457

Vranic, V., Menkyna, R., Bebjak, M., & Dolog, P. (2009). Aspect-oriented change realizations and their interaction. *e-Informatica. Software Engineering Journal, 3*(1), 43–58.

Wang, M., Lin, Z., Zou, Y., & Xie, B. (2019, November). Cora: Decomposing and describing tangled code changes for reviewer. In *44th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 1050-1061). IEEE. doi:10.1109/ASE.2019.00101

Williamson, B. (2018). The hidden architecture of higher education: Building a big data infrastructure for the ‘smarter university’. *International Journal of Educational Technology in Higher Education, 15*(1), 1–26. doi:10.1186/s41239-018-0094-1

Wlodarski, R., Poniszewska-Marańda, A., & Falleri, J. R. (2022). Impact of software development processes on the outcomes of student computing projects: A tale of two universities. *Information and Software Technology, 144*, 106787. doi:10.1016/j.infsof.2021.106787

Xiaodong, Y., Hiajun, C., & Guoliang, Z. (1998). Two-dimensional software development model combining object-oriented method with formal method. *Software Engineering Notes, 23*(1), 81–85. doi:10.1145/272263.272284

Xu, H., Cui, B., & Chen, C. (2021). Fuzzing with multi-dimensional control of mutation strategy. In *International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing* (pp. 276-284). Springer. doi:10.1007/978-3-030-79728-7_27

Zhang, L., Zhao, K., & Ni, Y. (2010, December). Concern framework supporting aspect-oriented software architecture design. In *2010 International Conference on Computational Intelligence and Software Engineering* (pp. 1-5). IEEE. doi:10.1109/CISE.2010.5676828
Meennapa Rukhiran is an assistant professor in information technology, Department of Computer Science at the Rajamangala University of Technology Tawan-OK, Chanthaburi Campus, Thailand. She received her Ph.D. degree in Information Technology from Rangsit University, Thailand. Her research interests lie in information technology applications, information retrieval, information services, and quality of information design. She has published international journals such as the Journal of Information and Communication Technology, TEM Journal, and Journal of Current Science and Technology.

Somchai Buaroong is a senior lecturer in the Business Computer department at Suansunantha University in Thailand. He earned a Master of Science in Computer Science from Rangsit University. He is currently pursuing a doctorate in information technology from Rangsit University in Thailand. His research interests include information modeling and frameworks.

Paniti Netinant is an associate dean of the graduate school and an associate professor in computer science at Rangsit University’s Department of Information Technology. He earned his M.S. and Ph.D. in Computer Science from Illinois Institute of Technology in the United States of America. His research interests include information modeling and frameworks, information technology design and development, the Internet of Things, and the layers and services of information. He has published in various international journals, including ACM Communications, ACM Computing Surveys, Journal of Information and Communication Technology, TEM Journal, and Journal of Current Science and Technology.