Journal of Current Science and Technology, May-August 2021
Copyright ©2018-2021, Rangsit University

Vol. 11 No. 2, pp. 240-260
ISSN 2630-0656 (Online)

Cite this article: Rukhiran, M., Netinant, P., & Elrad, T. (2021, May). 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

Journal of Current Science and Technology

Journal homepage: https://jcst.rsu.ac.th

# Multiconcerns circuit component diagram apply to improve on software development: Empirical study of house bookkeeping mobile software

Meennapa Rukhiran[1], Paniti Netinant[2,*], and Tzilla Elrad[3]

[1]Department of Social Technology, Ragamangala University of Technology Tawan-OK, Chanthaburi 22210, Thailand
[2]College of Digital Innovation Technology, Rangsit University, Bangkok 12000, Thailand
3Concurrent Programming Research Group, Illinois Institute of Technology, IL, Chicago, 60616, USA
[1]Email: meennapa_ru@rmutto.ac.th; [2,3]Email: paniti.n@rsu.ac.th; [3]Email: elrad@iit.edu

*Corresponding author; paniti.n@rsu.ac.th

Received 10 January 2021; Revised 10 April 2021; Accepted 29 April 2021;
Published online 27 May 2021

## Abstract

Developing component-based software is a demanding profession for software engineers. Developing component-based software is more complicated and needs more skills to meet software qualities, especially for mobile software design and development. Not only does mobile software have many platforms, but also a separation of concerns is required in the primary design and development, making the final component software very satisfying and comfortable to use the application. Since many prototypes have been redesigned and developed in a software life cycle, a prototype must undergo many components with multilayered and prime to duplicate components. The redundant components maybe failed to support the effective reuse because the components are contained several details and specifics. The house bookkeeping software can be decomposed into many components. The interaction and overlap of components are influenced by behaviors and classes. The limits in collaborations of reusable components can be found. In this paper, the multiconcerns circuit component diagram is originally proposed to express the development of component-based software, especially decreasing interactions of resemblance components. This technique uses a software component reduction between interactions of inter analogous inputs and outputs components, reducing a few redundant information, complex interactions, and tangling components. The result of the development makes software engineers better comprehend the design and implementation of component-based software proficiently. The empirical study of the house bookkeeping mobile software has shown an improvement of a component reduction in the final prototype of 26.47 percent over the previous technique using only an information flow diagram.

***Keywords***: *multiconcerns; circuit component; diagram; layering; software development; mobile software*

## 1. Introduction

Many researchers have approached a separation of concerns as a software adaptation to improve software quality attributes of a software design and development such as modularity, compensability, and reusability. The separation of concerns is to design the individual matter to support programming (Sommerville, 2014), such as class, method, procedure, etc. The separating concerns deliver the principal designing and programming paradigms of an aspect-oriented approach using weaving instead of calling the functionalities directly. The approach enhances a more reusable, extensile, and adaptable system (Diaz, Romero, Rubio, Soler, & Troya, 2005; Pinciroli, Justo, & Forradellas, 2020). The modularization is improved by generating new constructions for the encapsulation of crosscutting concerns into single modules named aspects and composing the crosscutting concerns together named weaver (Muck~~,~~ & Frohlich, 2014).

Software engineers can rapidly change among frameworks, behaviors, interfaces, and platforms for upgrade and maintenance with different techniques. Component-based software (Tibermacine, Sadou, Dony, & Fabresse, 2011) introduces a construction of software and concept reuses. The purpose of a component provides standalone services that design a specific architectural style. The component-based approach of software developments focuses on software requirements, architectures, designs, verification, formations, and distributions (Buhnova, et al 2014). The researches and development of component-based systems have been focused on many areas. Verma (2002) compares component-based software engineering with traditional software engineering and represents the benefit of reducing cost, decreasing development time, increasing term quality control (characteristic, performance, reliability, and usability), and increasing applicability of functional theories. Many researchers have designed component-based software supporting object-oriented development on web applications (Okewu~~,~~ & Daramola, 2014) and mobile applications (Giedrimas~~,~~ & Omanovic, 2015). A component provides core functionalities that can be implemented by coding and programming for several system requirements and user's requirements.

Moreover, components are designed for software development as architecture constraints that any features can parameterize. The business components are possible to support more complex or higher levels of software constraints. However, traditional software engineering, like the object-oriented approach, cannot reuse components significantly (Sommerville, 2014) and does not improve the system's internal design. The object orientation may fail to support the effective reuse because the single class contains several details and specifics. Behaviors and classes influence the interaction and overlap. Developers can find the limits in collaborations of reusable classes. An example of software solutions and aspect-oriented framework on the component-based software is proposed by Lee and Bae (2004). The technique has merged the reuse solution supporting separation of concerns on non-functional aspects (inter-component non-functionality). The component-based software enables the promotion of the reusability of components and connectors (Panunzio~~,~~ & Vardanega, 2014).

Our recent works are designed for housekeeping software based on three dimensions of layering (Rukhiran~~,~~ & Netinant, 2020a). The layering is separated relatively among datasets. The layers consist of concerns (income, expenditure, and liability) linking the X, Y, and Z axis coordinate. The functional data combination is from the three-dimensional layering of datasets that information can perform between layers. We have first applied the principles of Aspect-Oriented Software Development (AOSD) to separate the software's functions explicitly. The aspect elements are the minor functions that are engaged from crosscutting other concerns in a core system. By crosscutting concerns, the aspect element must not comprise during producer processes, and the developing software becomes scattered (duplication) and tangled (dependency) (Kumar, Kumar, & Iyyappan, 2016; Pinciroli, Justo, & Forradellas, 2020). The aspect elements have been excluded from the functional data. The execution design of an aspect-oriented approach through a weaver is proposed. One solution of our recent samples is a transfer component on the liability payment that can reduce the total of the liability data, decrease the income data by weaving on three layers (day, month, and year) and raise the expenditure data. A multilayered approach is provided for adapting a variety of crosscutting concerns. However, the design and implementation studies have not been completely designed yet. The previous research proposed Information Flow Diagram (IFD) with the Rapid Application Development (RAD) methodology (Chomngern~~,~~ & Netinant, 2017; Rukhiran~~,~~ & Netinant, 2020b).

In addition, software development was lack of the practical consequence of successful user capabilities in the deployment. The final software could guarantee to be misappropriated for users. In this research, we present the concept of a dimensional hyperspace through the functional data, describe the types of aspect elements, and design the execution flow diagram, which is one of the main concerns in the system usage level from the functionalities abilities of the house bookkeeping software. To state the important diagram, we also propose a Multiconcerns Circuit Component Diagram (MCCD) of component-based software. The MCCD is the ease of use by dividing the functional patterns of contracts (sets of data in layering and separating concerns) and connecting components from inputs and outputs. A concern circuit is a pattern of structures using a principle design of a circuit that characterizes a logical relation of component-based software in the development. Our proposal aims to design the execution flow of the component and the concerned circuit and prove the execution through the vertical and horizontal layering in the implementation phase. We have proposed the conceptual framework design of the MCCD diagram in the case study of the house bookkeeping software design. By demonstrating the empirical components, the concept of the MCCD can be applied adaptably competent in application software designs.

## 2. Objectives and research questions

This research's main objectives are primarily to design and implement component-based software like a house bookkeeping mobile application for users' diverse technological skills and suitably better deliver the final software product of the house bookkeeping mobile application. The succeeding research questions are defined to comprehend the objectives.

RQ1: This is the technique to aid the design and implementation of component software to support the Information Flow Diagram. Is the technique suitable for developing component-based software development like a house bookkeeping mobile application?

RQ2: How can Multiconcerns Circuit Diagram better comprehend the design and implementation of component-based software efficiently?

According to an agile software development system named RAD (Rapid Application Development), the model arranges rapid prototype releases based on users' iterations and satisfactions. However, the designers and developers cannot promise the final product met better design and deployments of variability component-based software. The first research question aims to study a better solution during the plan, design, and development phases. The MCCD methodology can agreeably express the whole system components and layers of component interactions with the support of multiconcerns. The second research question is to evidence the better design and development of component-based software development. Our final production of the house bookkeeping mobile application has used this methodology in the deployment.

## 3. Literature review

The research study establishes software design and development using circuit component design of the house bookkeeping mobile software. Accordingly, many studies have been provided in this research review.

### 3.1 Separation of concerns

Separation of concerns is defined as a critical principle of software designs and implementations. A concern is divided as a part of the software that represents a single functionality. The aspect-orientation is an approach to handling the separation of concern through new abstractions and composition mechanisms (Muck~~,~~ & Frohlich, 2014). The design principle of aspect orientation software is to augment crosscutting concerns' modularization (Tanter, Figueroa, & Tabaerau, 2014; AI-Hudhud, 2015). The concerns can be called by the components depending on a weaver. Weaving is a process of systematizing aspects and other elements (Zhang, Khedri, & Jaskolka, 2012; Lindstrom, Offutt, Sundmark, Andler, & Pettersson, 2017). The evolution strategies of aspect-oriented software focus on defining the four rules by explaining the event, condition, and action for supporting the changing of computation environment (Zhang~~,~~ & Rong, 2009). The dynamic evolution is concerned with a running time. The first rule is an addition of a base component. The second rule is an addition of an aspect component. The third rule is an addition of an aspect connector. The fourth rule is the addition of attachments. Therefore, the separation of concerns can result in reusable, extensile, and adaptable systems.

### 3.2 Component-based software

Component-based software is defined as an architecture constraint to validate the specific architectural elements (components) (Tibermacine, Sadou, Dony, & Fabresse, 2011). A study of aspect-oriented software architectures for code mobility is composed of components (Lobato, Garcia, Romanovsky, & Lucena, 2008) and aspects. The aspect is represented using the symbol of a diamond shape, and the crosscutting interface is displayed using a small grey circle with its name placed over the circle. The separation of concerns can solve the fine-grained problem. The architecture becomes a clean modularization, an explicit introduction, and an improving variability of programming with flexible incorporation of code mobility. To prove the consistency of components, a system's intra-component dependency models enable a determination of the dynamic adaptation (Sadeghi, Esfahani, & Malek, 2017). Hoffman and Eugster (2008) mention that aspects' ability is the separation of concerns, and the modularization is transformed into reusable components. Creating explicit modeling crosscutting concerns and an appropriate aspect-oriented technique can achieve the semantic separation of concerns. Design aims are to reduce coupling and decrease cohesion by counting the number of modules explicitly named point cut. The pointcut is defined as a state of selecting the specific joint points. Component-based development (CBD) has been designed for supporting the encapsulation of collaborative behaviors crossing multiple components through the explicit architectural element. The connector enables to contain and reveal gross structures and global control flows, including the entire system's behaviors such as design decisions, collaborative protocols, and functions incorporated into the non-functional aspects using connectors (Lee~~,~~ & Bae, 2004).

3.3 Early stage of our house bookkeeping software design approach through Aspect-Oriented Approach (AOA)

Our recent Aspect-Oriented Software Development (AOSD) is designed for supporting the house bookkeeping software by separating the functional data and the aspect elements. In our recent work, we have proposed the functional data through a three-dimensional layering to present the relationships between sets of data and dimensions. There are three dimensions (incomes, expenditures, and liabilities) dividing from a series of data concerns. Each dimension is categorized into smaller datasets shown in our latest work (Rukhiran~~,~~ & Netinant, 2020a). The functional data initially recorded from one field to n fields in table names. The aspect elements define as a set of computational properties (e.g., insert, update, delete, day, month, year, and sum) which starts corporately more than one aspect to m aspects. An object executes calling the aspect elements and the functional data using crosscutting concern in an upper level. We assume a weaver to call the object for the final execution by using the functional formula n x m for crosscutting concerns, as shown in Figure 1. Weaving is the process of transforming to solve scattered solutions and avoid tangled methodology.

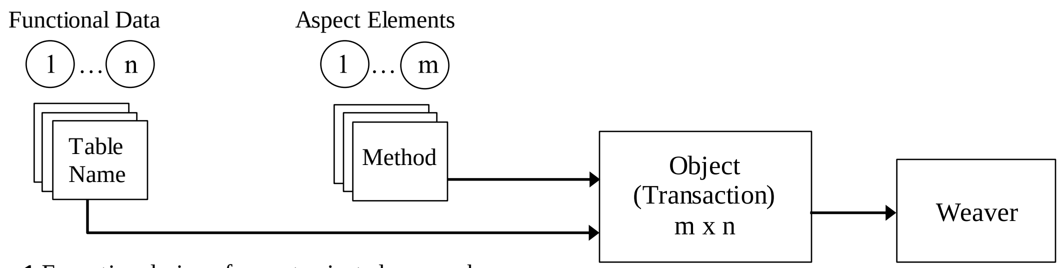

**Figure 1** Execution design of aspect-oriented approach

A glass of iced black coffee (Americano) can explain the execution of AODA. Coffee is a functional data, and a cube of ice is an aspect. Hence, a glass contains many cubes of ice like our design. We separate the computational functionalities into a single methodology. Our weaving process performs a transformation of crosscutting by advising at running time logics. The dynamic transition of aspects is a process that can return a calling operation to the object without any effect at a compiling time and a running time.

On the other hand, a blending coffee is a mixture of ingredients that we cannot get any ice like an Object-Oriented Design Approach (OODA). OODA focuses on representing problems using objects and their behavior. An OODA implementation leads to code scattering and code tangling (Gupta, Singh, & Kumar, 2016), while AODA deals with breaking down the methodologies using the separation of crosscutting concerns. AODA also leads to an increase in the modularity of components and reusability of aspects.

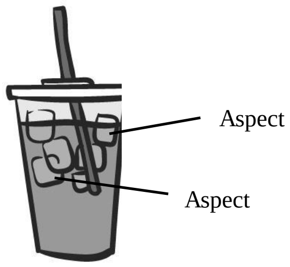

**Figure 2(a)** Comparison of AODA using black coffee

**Figure 2(b)** Comparison of OODA using blend coffee

3.4 Adaptability of the framework in software design

The principal key to achieving adaptability in software design is compatibility and interoperability. These elements are necessary for a rapidly changing environment. The adaptive application in QoS (Quality of Service) is to interact with the implementation of changing resource conditions (Witana, Fry, & Antoniades, 1999). Software services need to be adaptable and aware of the contexts for experiencing the best QoS. Self-contained adaptable applications provide naturally dynamic adaptability in the environment by reacting at a running time. The formal model is to specify the characteristics with aspects of applications and environments managed by a framework. The framework for context-aware, adaptable software applications and services is conducted by Benedetto (2011). The user's QoS requests on an an-hoc client application. The adaptable application development is carried out as a routine engineering activity, and it brings an independent evolution from the other tasks. The strategy designing pattern, specialization classes, adaptation classes, and Objective-C categories specification are produced to QoS approach. Kebir (2012) has proposed a combining approach called JACAC basing on aspects and components to enable the autonomic capabilities in the self-adaptive software system. Dynamic reconfiguration is one of the solutions that a framework can implement through an object orientation. The new component basing on a software system can be applied when the new services or functionalities are replaced and adapted in either the functional or non-functional dimension. An extensible framework is implemented for identifying the aspect-oriented refactoring opportunities on an extractor. The architecture consists of layers to support the refactoring of the extensions and the interaction with users through a pluggable architecture (Boukraa, Boussaid, Bentayeb, & Zegour, 2013).

## 4. Our house bookkeeping software framework design

According to our early execution design of house bookkeeping software in Figure 1, two main fundamental concerns are separated using aspect orientation: aspect elements and three-dimensional layering called functional data. The object can use the composition of these two main concerns. In this practical programming, the object is known as the component. The component is software constraints called aspect elements and functional data to execute in a compiling time. Application software may consist of many components. In this section, we have divided our separation of software concerns into four sections. Section 4.1 clearly understands aspect elements of house bookkeeping software design and how it works with components. Section 4.2 delivers a basic crosscutting concept of three-dimensional layering and aspect element layering through a hyperspace design. Section 4.3 presents our early designs of the component-based approach. Section 4.4 offers an adaption of our multiconcerns circuit component diagram. The multiconcerns circuit component is an extension series of our components from section 3.

4.1 Principles of aspect elements

4.1.1 Aspect element declaration

For designing software, functional and non-functional requirements decompose the functionality of a system. The functionality can support many components' encapsulation, and each component may require many objects to collaborate in an operation. An object's explicit design is separated from the functional data and the aspect elements in our design. In this context, we only provide the functional aspect named aspect element. The definition of the aspect element is a set of crosscutting properties that tangles in the system's functionalities or methodologies. For instance, the aspect elements in house bookkeeping software design, the thirteenth functionalities of the aspect elements are designed separately in Figure 3. Some more aspects are presenting in the following sections. The aspectual properties are excluded from the functional data.

4.1.2 Type of aspect element and how it works

By defining each aspect's functionality, we have fundamentally realized that the framework can declare the aspects into two different types. Each aspect has well-defined roles as follows:

1. Individual aspects: The individual aspects work once when the object is called. However, there is more than one aspect that can be called to execute at the joining point.

- Insert: The insert aspect is called when the transaction has requested to insert data. The object or the transaction will call the insert function once, simultaneously, and the object also calls the functional data. The execution depends on the object that is called on.

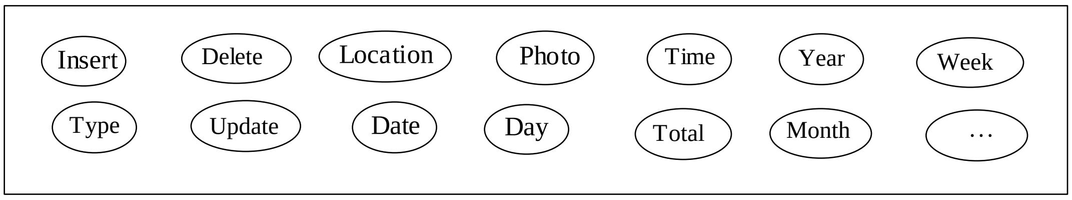

**Figure 3** Aspect elements of house bookkeeping software design

- Delete: The delete aspect is called by the object when the weaver is requested. The execution works completely when the functional data is also called to send the correct concern. The delete function will be accomplished when the object receives the transaction from the aspects and the functional data.
- Update: The update aspect is called when the weaver is reached through the display component. The object will request the aspects and functional data that depend on the requesting object's purpose. In Figure 4, the transaction containing the update aspect and other aspects, such as a type, a location, and a photo. We assume the first transaction of crosscutting between aspects as t1. The object is assigning as a weaver of t1 support many crosscutting points.

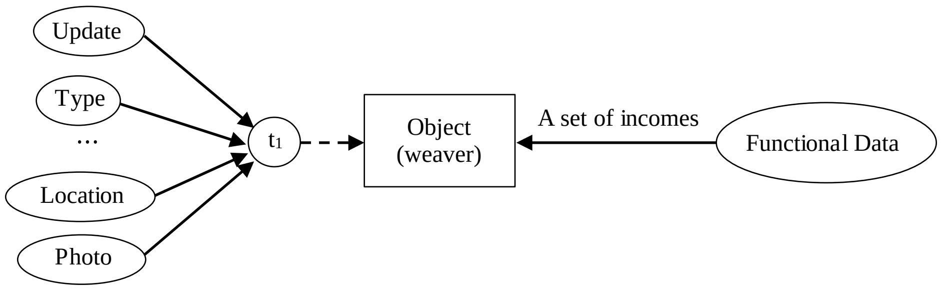

**Figure 4** Execution of update function of income

- Notify: the notify aspect can be requested when the due date of a payment is set up for reminding.
- Location: The location aspect relating to the space stamp is assigned as a function of the current location of spending or receiving money. The location function cannot be called individually. It needs to be requested through the other aspects like an insert, an update, and a delete aspect.
- Photo: the picture aspect is an optional function that can be called to request proof of payment. The advice works as an analyser to identify the aspects that a photo can select.

2. Persevere aspects: an arithmetic sequence can explain the continuous calls of aspects. We assign the first transaction of the sequence as t1. N is a number of transactions. Tn is the term of the last transaction. Therefore, the formulation of the persevere aspects can be given by t1 to tn where $n \geq 1$. For every sort of a

countable infinite is {t1, t2, t3, …, tn}. Each transaction can be called at the same time. It depends on the requests from the object.

- Date: The dating aspect is called when the transaction is requested. Weaver can reach the aspects directly. For example, a tracking operation (a component) of a financial statement is asked for through the object from a display operation (a component) by combining the date 1 to date 5 of the recordings. The object will call on the functional data to request displaying three-dimensional layering (incomes, expenditures, and liabilities). The date aspects are also asked by calling the 1st date to the 5th date, as shown in Figure 5.
- Week: The week aspect is called by specifying the week function. A transaction can work the execution collecting a transaction through the object for seven days. This design is suitable for calculating an account's total and displaying its data using a scheduled task.
- Month: The month aspect is the function of calling the transaction for a month. The aspect has a concept design like the week aspect. Collecting different concerns can reach each aspect that depends on the functional data.
- Year: The year aspect also has the same purpose as the date and week aspect, but the execution can be called covering an amount of dataset in a year.
- Time: The object calls the time aspect. The function can return the time value.
- Type: The type aspect is specially designed to support the categories of the functional data. The aspect can be called corporately to reach different types of datasets.
- Sum: The sum aspect is a function for calculating several transactions. In Figure 5, more than one aspect can contain a transaction. Each aspect is identified using a number as series to summarize the final result.

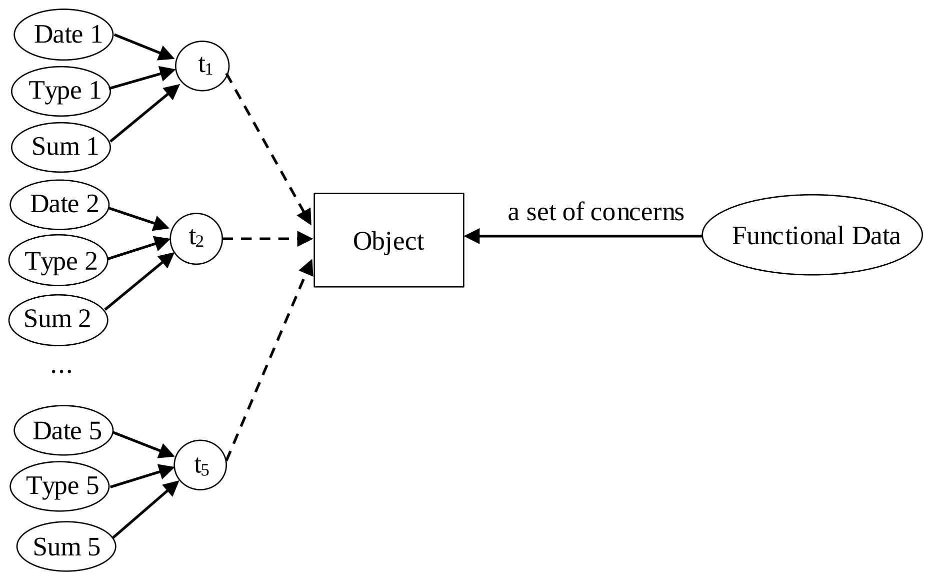

**Figure 5** Execution of date aspect

Both individual and persevere aspects can be called cooperatively in an execution. The operation of inserting an expenditure account works through the object. There are many aspects (e.g., insert, date 1, time 1, location, and …) and an expenditure concern (a concern in the functional data) called by the object. Moreover, the optional process can work causally by inserting a notification to remind regular incomes and payments. The object also calls date and time aspects that can be set up for a scheduled notification, as shown in Figure 6. Then the object will call the weaver for execution at a running time.

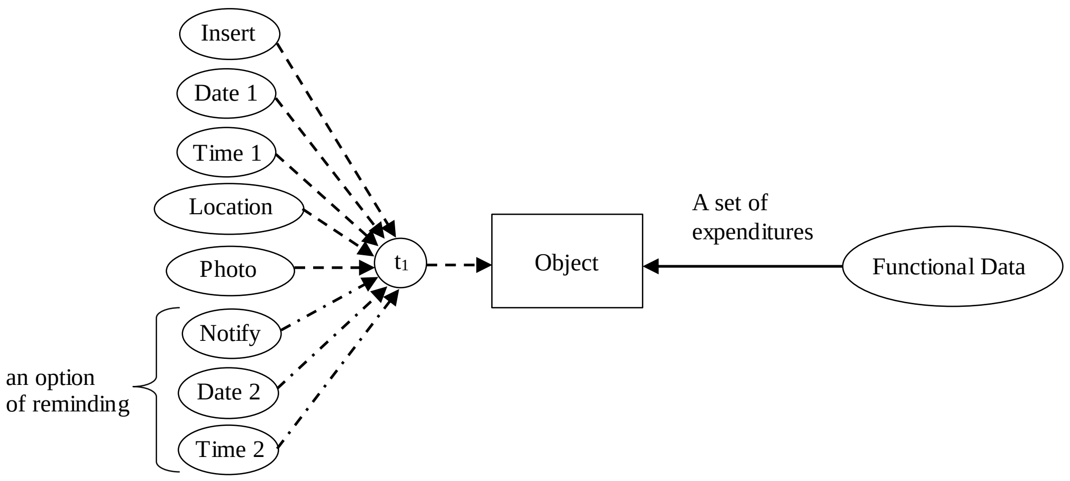

**Figure 6** Execution of insert function of expenditure

The overview execution of separating concerns is illustrated in Figure 1, and the composition of using the aspect elements and the functional data by an object has represented in this section. However, the explanation does not include how an object works with another object. In our design, we assign the object a component that can be defined to manipulate the functional data and the aspect elements. Therefore, many components should provide regarding the principle design of the separation of concerns. The components of the framework will present in the next section.

4.2 Crosscutting concept of hyperspace design

Crosscutting point is required to execute the functional data and the aspects using intersections of a single crosscutting point. Each dimension cuts across a single element aspectual property to implement relatively because of crosscutting concerns shown in Figure 7.

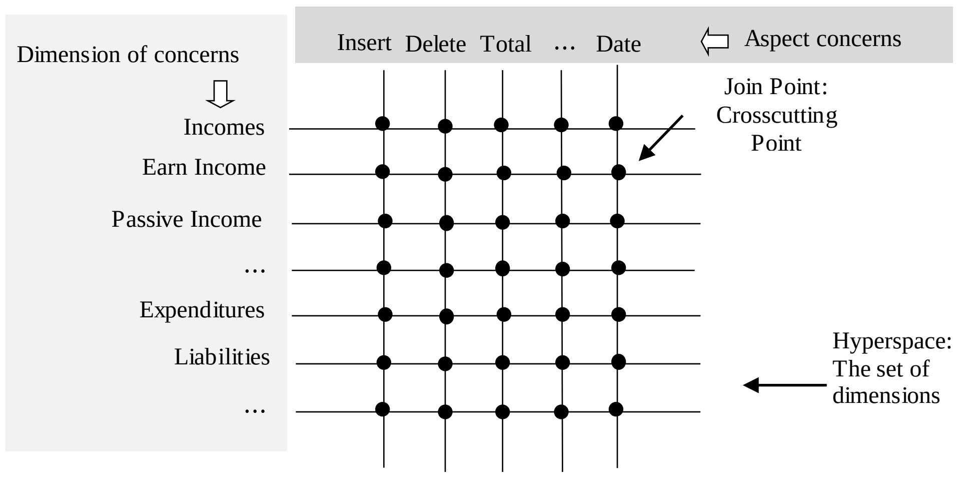

**Figure 7** Hyperspace design of functional data crosscuts aspect elements

The separation of concerns is able to be potentially used crosscutting the different requirements. The process integrates the functional data and the crosscutting concerns called weaving. When a transaction is reached, the object associated with a method call will analyze the transaction. The different invocation of this methodology is presented dynamically and flexibly. It depends on the aspects as well as the functional data that are invoked in the execution. The functional data is independently used and represented the dimensions that are associated with a method call. The weaver can affect the different datasets from the dimensions. The aspect can be called depending on the execution at that time. The three dimensions present a set of dimensions in a hyperspace approach. The hyperspace approach defines a space of the multi-dimensional matrix by containing a specific concern of a dimension. The hyperspace of this design is a concerned space in the three-dimensional design that adapts the multi-aspect. A concern space consists of a set of concerns and components. For example, a one-dimensional layering may require for getting datasets from an axis. An x-axis consists of datasets from an expenditure concern, a y-axis only presents datasets from an income aspect, and z-axis is datasets from a liability concern. Two-dimensional layering shows the relationship between two axes by combining two layers. The ability of the hyperspace in our design achieves the dynamic effects. Moreover, a new concern and multi-dimensional dataset can be incrementally proposed (Khanzadi, Shahbazi1, Arashpour, Ghosh, & 2019).

### 4.3 Component-based approach of house bookkeeping software

#### 4.3.1 Software requirements

The development of house bookkeeping software at the system level is constructed to cover a personal finance performance. The scope of the software should be focused on functionalities and abilities as follow:

1. Set up for the first usage, such as selecting recording accounts of income, expenditure, or/and liability and manipulating the categories of each recording and the risk condition of financial statements. For example, the function will notify when the amount of expenditures and liabilities is more significant than income.
2. Record transactions of an income, expenditure, and liability daily by categorizing the separation of concerns. Smaller categories are dividing from sub-dimension. The software also supports using infographics of each item.
3. Adapt a financial statement for the day, week, and month using aspect elements' concept design.
4. Track for inputting everyday transactions and payments automatically. For example, a user may set the notification to record an amount of money to spend on lunch.
5. Notify for a chronological payment date and a user configuration of financial finance. For example, the system will notify the payment due date monthly of a credit card.
6. Manage liabilities by transferring a recording of a liability dataset to an expenditure dataset and reducing an income dataset. For example, a user makes a payment of a credit card. The user also has to record the payment, and the software will transfer a transaction by increasing the amount of expenditure and decreasing the amount of income.
7. Backup data for synchronizing files to a local database and cloud computing, such as uploading the dataset transactions.

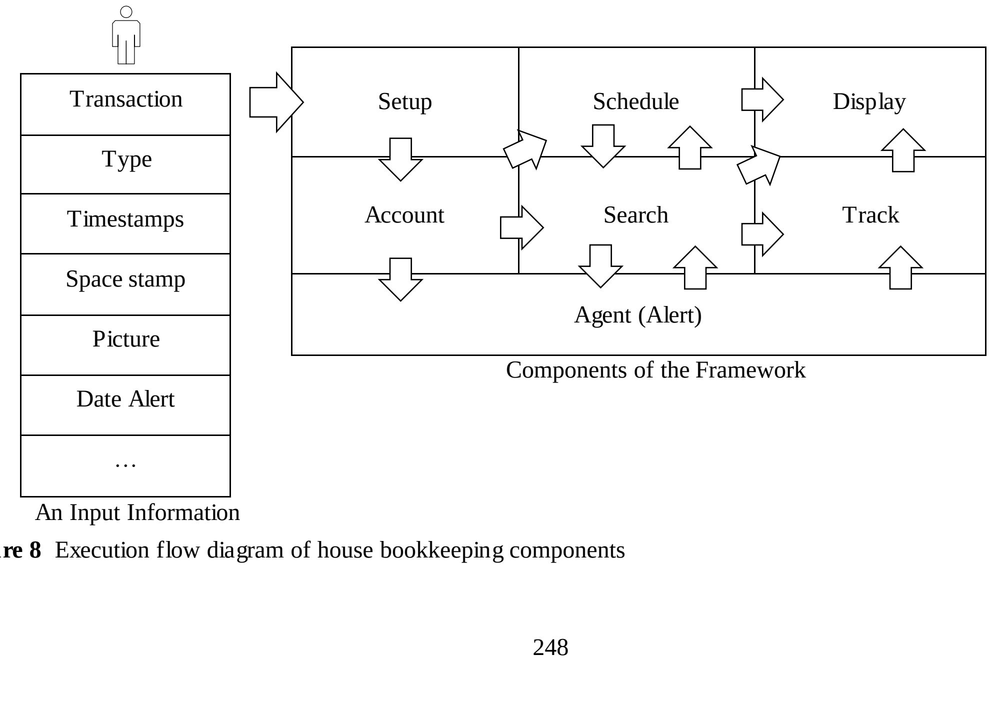

**Figure 8** Execution flow diagram of house bookkeeping components

4.3.2 Software component-based declaration and execution

Figure 8 shows that the framework's component is a component-based process of the execution flow in a running time. The component represents the software-defined steps that emphasized the design phase for constructing software in the implementation phase. The transaction flow diagram can use independently for the development of application software in any platform language. There are seven components (setup, account, search, schedule, track, display, and agent intelligent) described in the execution flow diagram. The components provide the structural sequences of an operation to control the execution and achieve the systematic reuse component. The input information on the left-hand side in Figure 9 contains the samples of data recordings transferred into the execution component flow. The input information consists of many data recordings. For example, a transaction is an amount of money, a type is a category of funds that is separated as the functional data, a timestamp is a sequence of characters of a specific event (e.g., date and time literals), a space stamp is an indicating particular location and the data of the location can be used Global Positioning System (GPS) coordinates to track the current location easily, a photo is a proof of payment that it can be related to the details of payment, and a date alert is a service to remind the payment for due date alert. The input information can make a different effect to support in a finite state machine. We have designed the execution sequences between the components in Table 1.

**Table 1** Samples of processes between components

| State Functionality of components | Description |
|---|---|
| Setup → Display | The software provides the first configuration, which contains user information, record domains, and risk conditions for a notification system. The event is to execute by transiting user configurations to the state of the account component. |
| Account → Search → Display | The state supports inputting transactions into an appropriate type for keeping data and transactions by separating data types and checking for an existing record. |
| Account → Search → Track → Display | The tracking state is executed corporately with the searching component. A dynamic search of data recordings is activated using the day aspect, the week aspect, the month aspect, and the year aspect to display finance reports. The software provides tracking adaptively on different timing and dynamic datasets in the three-dimensional layering. |
| Account → Schedule → Display | A financial statement is monitored by planning due payments and usual spending and receiving money. The schedule component approaches a database to query a due date for reminding payments and send a push notification for alerting a user to record a usual transaction. |
| Account → Agent → Track → Display | The agent component is a future stage of gathering data from databases kept recording by the account component. The application of data mining techniques and algorithms should use to analyze and forecast for improved financial management performance. The different agents (an active agent: keep tracking all times and a passive agent: execute by requiring or setting an alert on) should be provided supporting an operating for monitoring data and alerting to an automatic notification. The tracking component's reminding system should include an ending and warning for a user's financial risk conditions. |

Moreover, the application software is divided into many components having their states. The component is a part of the abstract system that it cannot execute individually. A component is an architectural element that is assigned as the main functionality to communicate with other components. Each component provides its services (method) separately, but the components are synchronized to others, depending on the system's requirements. For a synthetic transaction of the separation of concerns, more than one component may require executing corporately to another, and more than one aspect enables to across in a component. We have represented the framework design components for supporting house bookkeeping software, and the transaction flow diagram is provided sustaining the execution flow of components. The weaver cooperates between components for crosscutting concerns the aspect element. To propose the conceptual framework components through the separation of concerns, we intend to clarify the processes, structures, and functions. First of all, the process is defined as a systematic sequence of actions that control the flow of data and the operations between the components providing procedures, status, and behaviors and tells a developer how the software works. Secondly, the structure has defined a boundary of each functional element. Thirdly, the function is defined as the role of the operations describing the capabilities of the software. We represent the circuit components' structures using the physical connection of circuits to prove the component-based software. By separating seven-circuit components, we define the processes, structures, and functions as follows:

- Setup Component is an initial component to provide the software configuration before starting the application usage. We have designed a circuit design of a circuit to represent a component circuit shown in Figure 9. The three types (income, expenditure, and liability dimensions) of data recordings arrange for a user to set up a user account. The categories of each type (subdimension) are available for separating records. The setup component consists of the functional data, the aspect elements, and the user configuration. Based on the user's requirements, the component can select the types of recording. A user may not establish a liability recording if he/she does not have a credit cardholder. After choosing some subcategory of incomes, a user may receive money from many sources like a salary and selling products online. Thus, the user can create the subcategory of revenues. The final selection of the notification on the user's financial statements, such as the user may set up the report of risk conditions when "an amount of expenditures is more significant than several incomes and is less than several liabilities. The statement can provide the symbol of I < E < L. Moreover, the user requests can often adapt the selections. We use the language of set theory to present the collection of those elements. We let the event where:

*A set of the setup component = { Functional data, Aspect, Configuration };*
*The functional data = { Incomes, Expenditures, Liabilities },*
*Aspect = { Create, Insert, Type, Save, ... },*
*Configuration = { Type of Records, Subcategory, Configuring Condition, ... }.*

- Account Component is a manipulative stage for supporting insert, delete, and update statements. It manages the activities of data inputs and records data. The majors of recording are income, expenditure, and liability dimensions. Moreover, a user can setup to remind a usual record by including another component for this execution. We let the event where:

*A set of the account component = { Functional data, Aspect, Configuration };*
*The functional data = { Incomes, Expenditures, Liabilities },*
*Aspect = { Create, Insert, Update, Save, Delete, ... },*
*Configuration = { Debit, Credit , ...}.*

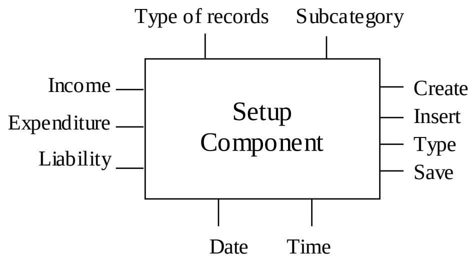

**Figure 9** Structure of a setup component

- Search: is a browsing stage by separating the categories of concerns. It enables the specific searching to support the different attributes for displaying the results through a graphic data report. The component is designed to support the relational and non-relational databases. The separating of concerns provides a useful technique in designing databases and querying data. We let the event where:

*A set of the search component = { Functional data, Aspect, Configuration };*
*The functional data = { Incomes, Expenditures, Liabilities },*
*Aspect = {Type, Date, Time, Day, Month, ... },*
*Configuration = { Select, Where , Group by, Having, ...}.*

- Schedule: A stage of arranging is for reminding of usual recordings and payment programs. The software must provide an input process of time reminding, and if the user setups for scheduling a due date of payments, many notification techniques should apply.

*A set of the search component = { Functional data, Aspect, Configuration };*

*The functional data = { Incomes, Expenditures, Liabilities },*
*Aspect = { Date, Time, Total, ... },*
*Configuration = { System, Email , SMS, MMS, ...}.*

- Tracking: A stage of reporting a use financial statement by tracking dynamic data. For instance, the conceptual design of aspects such as a day aspect and a week aspect allows users to track their finances at different timing.

*A set of the search component = { Functional data, Aspect, Configuration };*
*The functional data = { Incomes, Expenditures, Liabilities },*
*Aspect = { Type, Total, Day, Month, Year, ... },*
*Configuration = { Graph, Number, ...}.*

- Display: A display component is the final stage of every component. Graphic User Interface (GUI) can represent connecting with other devices like a printer, a monitor, and a voice (Text to Speech).

*A set of the search component = { Functional data, Aspect, Configuration };*
*The functional data = { Incomes, Expenditures, Liabilities },*
*Aspect = { Type, Date, Time, Day, Week, Month, ... },*
*Configuration = { Monitor, Printer, Speech, ...}.*

- Agent Intelligence: An agent component is the stage of an intelligent system. An innovative housekeeping software is a challenging key to integrating the advantaged techniques such as data mining and data scientist to analyze and forecast trends in personal finance from the existing data. For example, predicting the most spending on a month's expenditures and suggesting a payment of interests and debts to reduce a cost. The framework is designed to support dynamic adaptability. For example, in the agent intelligence component, an existing application can insert components without any effect at runtime.

### 4.4 Adaption of multiconcerns circuit component diagram

We have illustrated the two samples of the component-based circuits for presenting the component-based software using circuits' design. The interconnections between components are shown by crosscutting of layers. Each component is generated differently in a multiconcerns circuit using Interface Abstraction Layer (IAL) to connect layers. A layer is widely applied to group architecture patterns in different levels of system abstractions (Gama~~,~~ & Donsez, 2011). A higher layer enables to call services on the neighboring lower layers with the providing aspects (Netinant~~,~~ & Elrad, 2016). LIA is a low-level perspective of separating multiconcerns circuits that allows independent concerns and data granularity. The overall objective of the design is to separate the multiconcerns circuit from the abstraction layer. We assume several setup circuits into three concerns (from one to three numbers): a circuit of an income dimension, an expenditure dimension, and a liability dimension, respectively. The setup circuit's sample design can fulfill specific roles and practical concepts, as in Figure 10.

In Figure 10, a setup circuit is the first important state for users' signing up before using a house bookkeeping application software. The circuit consists of the functional data (income, expenditure, and liability), the aspect elements (day, week, month, and year), and the configuring conditions for applying financial statements (e.g., $E < L < I$, and $I < E < L$). Because the analyzing phase, which is one stage of the Software Development Life Cycle (SDLC), is based on the user's requirements and adoption of information technology to build powerful and efficient software to support personal finance management and an economy. For creating an account, a user must choose record types for managing the personal account (income, expenditure, and/or liability recordings) and a notification of the risk conditions of financial statements. We use the mathematical symbols: greater than (<) and less than (>) to compare the money flow. For example, the user may set up the notification system when "an amount of expenditures is greater than several incomes and is less than several liabilities." The statement can be proved by the symbol of $I < E < L$. On the other hand, the designing phase should provide an account generation's capabilities and the notification by considering the component that the action is called and the configuring condition is stored in a database. Moreover, the functional data is design using three-dimensional layering to provide a more loosely coupled software design and high cohesion. The separation of concerns handles the reusable software components.

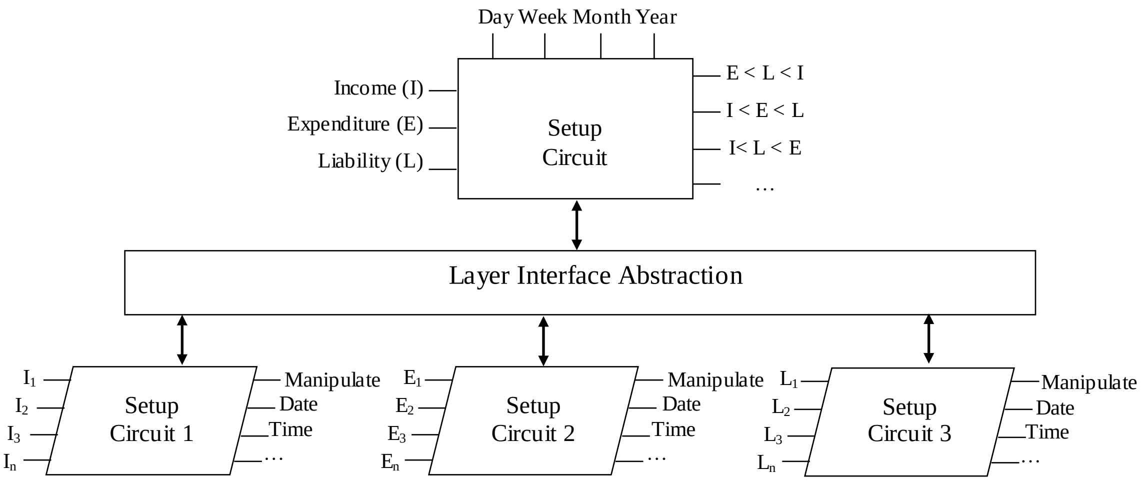

**Figure 10** Execution flow diagram of setup circuit

Layer Interface Abstraction (LIA) provides a set of multiconcerns circuit components. A circuit comprises three-dimensional data and the aspect elements (e.g., insert, delete, update, time, date). There are three numbers of accounting circuits that belong to different datasets. The data recordings of accounting circuit numbers one to three are a set of incomes, a set of expenditures, and liabilities. A set of data in one dimension on one layer is a set of incomes (I), expenditures (E), and liability (L). The component-based circuit does not take place to be executed respectively. The execution depending on the method call, is requested adaptively using a weaver. The formal notation of one-dimensional = {{I}, {E}, {L}}. For example, the set of incomes must start from the first record to n record. We express the set of incomes = (I1, I2, I3, …, In) on one layer. The formal notation of one-dimensional = {{I}, {E}, {L}}. A set of data in two dimensions on one layer is a Cartesian product of sets. Each set of the dimensions intersects for all. Thus, there are three formal notations of two-dimensional = {{ I, E }, {I, L}, {E, L}}. For example, the cross product of I and E is denoted by I x E. We set I x E = {(i,e) | i $\in$ I and e $\in$ E}. A set of data in three dimensions on one layer is also a Cartesian product of sets. The formal notation of three-dimensional = {{I, E, L}}. Consequently, to manipulate more than one recording of data, we design LIA as a layer to crosscut among circuits between the functional data and the aspect elements through the weaver. The dynamic weaving can integrate adding and removing between concerns and components at a compiling time and a running time.

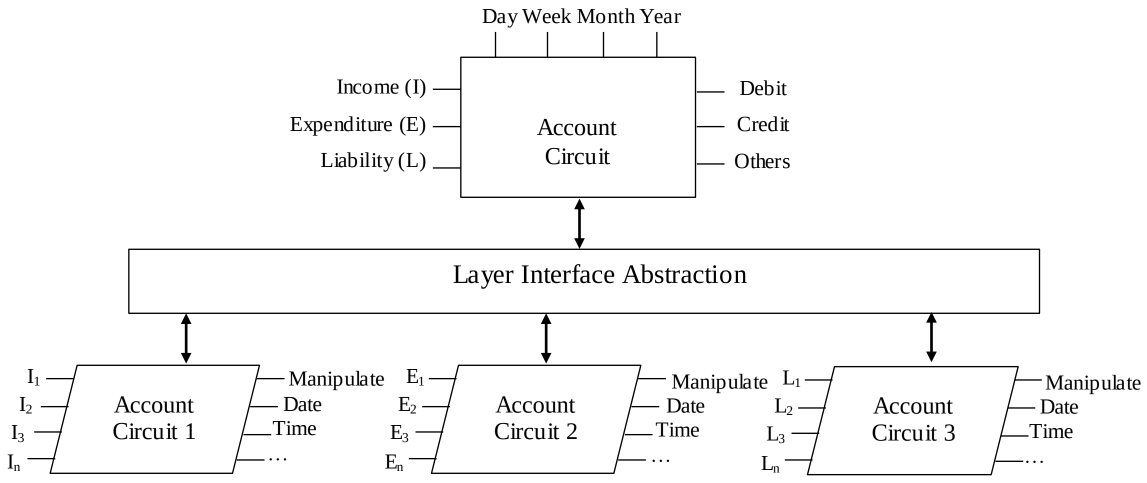

**Figure 11** Execution flow diagram of account circuit

Figure 11 shows another sample of an account circuit presenting through the functional data and the aspect elements. It illustrates the functional relationship of data manipulation (an insert, an update, a delete) through the configuration of debit, credit, and others. A debit is an execution of increasing an amount in the payable account. A credit is an execution of increasing a negative amount and decreasing an amount of money paid from an account. Because the desired features in the software design should focus on user behaviors, the others of unknown and forgotten transactions can exist because a user may not remember where the money comes from and spends on. Thus, we name these anonymous records as others.

## 5. Evaluation

5.1 Operational semantics of using three-dimensional layering

A new idea of dimensional layering is used to describe the sets of data dividing into three concerns. The data sets' composition can represent the dataset of house bookkeeping for executing data in the component-based circuit. The axis layering is provided three different semantics containing an income layer, an expenditure layer, and a liability layer, as shown in Figure 10, 11. The layer provides the appropriate contextual information for data manipulation. Each dimension consists of a set of multilayers such as a y-axis of an income layering, Income = {I1, I2, I3, …, In}, refers to one layering of sub-dimension which is divided from user's data categories (e.g., a passive income and an earn income). There are two types of quantifiers using to express the formal notations for computing the functional data from datasets. The universal quantifier (∀) is for a selection of all records from a layering. We describe as ∀income, income > 1. The existential quantifier (∃) is for some records in the universe. We express as ∃income, income > 1. The quantifiers can also be used to express through the two-dimensional for two layerings and more. For example, selecting a searching component is compared between all categories of incomes and some categories of expenditure. We set the sample of two layerings as ∀income U ∃expenditure.

The dataset of an income layering from one to one horizontal or vertical or oblique line can take to execute with some aspects through the weaver. A combination of the functional data is from a set of data between layers. A transformation of weaving includes the functional data and the aspect elements which cut across from the method call. For instance, the layers of an income dimension are computed to show several salary categories in March 2017. The transformation of weaving must execute through the functional data of an income layering. The aspects are the type aspect, the total aspect, the month aspect, and the year aspect, as shown in Figure 12 at point number 1 (1). We let the type = {$\text{Income}_{\text{salary}}$}, the total = {sum():∑ n}, the month = {May}, the year = {2017}. We assign a symbol of crosscutting concern by a weaver $\otimes$ . For each execution, an amount of salary categories on March 2017, we compute

$$\exists Income_{salary} \otimes Total\ U\ (Month,\ Month = May)\ U\ (Year,\ Year = 2017) \quad \text{or}$$
$$\textstyle\sum^{n}_{\exists income_{salary}\, \in\, Income} \otimes (May\ U\ 2017).$$

However, at the same point an amount of salary may show on May 2016, it depends on the method call of the functional data and the aspects as at (2). The aspects are composited relatively, but the same aspect may be showed different semantics depending on parameters. We let the type = {$\text{Income}_{\text{salary}}$}, the total = {sum(): ∑ n}, the month = {May}, the year = {2016}. By computing, the statement can be assigned

$$\exists Income_{salary} \otimes Total\ U\ (Month,\ Month = May)\ U\ (Year,\ Year = 2016).$$

The formula is expressed using a composition of two layers, such as a cutting point from one horizontal and vertical layer to two layerings, to compute relatively between two dimensions. For instance, in (3), the financial statement has computed a balance of incomes and expenditures from 1st – 15th March 2018. We set the type = {Income, Expenditure}, the total = {sum(): ∑ n}, The day = { 1, 2, 3, … , 15}, the month = {March}, the year = {2018}. We express

$$(\exists Income \otimes Total\ U\ (Day,\ Day = \{1,2,3,\ldots,15\})\ U\ (Month,\ Month = March)\ U\ (Year,\ Year = 2018))\ U\ (\exists Expenditure \otimes Total\ U\ (Day,\ Day = \{1,2,3,\ldots,15\})\ U\ (Month,\ Month = March)\ U\ (Year,\ Year = 2018))$$

or

$$(\exists Income\ U\ \exists Expenditure) \otimes Total\ U\ (Day,\ Day = \{1,2,3,\ldots,15\})\ U\ (Month,\ Month = March)\ U\ (Year,\ Year = 2017))$$

The three layerings are designed to support the computation of three dimensions for showing an amount of income, expenditures, and liabilities. In (4), the relation of a cutting point is called from one horizontal, vertical and oblique line. Comparing three domains can represent a balance of incomes, expenditures, and liabilities in June 2018. We let the type = {Income, Expenditure, liability}, the total = {sum(): $\sum$n}, the month = {June}, the year = {2018}. We compute

$$(\exists Income \otimes Total\ U\ (Month,\ Month = June)\ U\ (Year,\ Year = 2018))\ U\ (\exists Expenditure \otimes Total\ U\ (Month,\ Month = June)\ U\ (Year,\ Year = 2018))\ U\ (\exists liability \otimes Total\ U\ (Month,\ Month = June)\ U\ (Year,\ Year = 2018))$$

or

$$(\exists Income\ U\ \exists Expenditure\ U\ \exists Liability) \otimes (Total\ U\ (Month,\ Month = June)\ U\ (Year,\ Year = 2018))$$

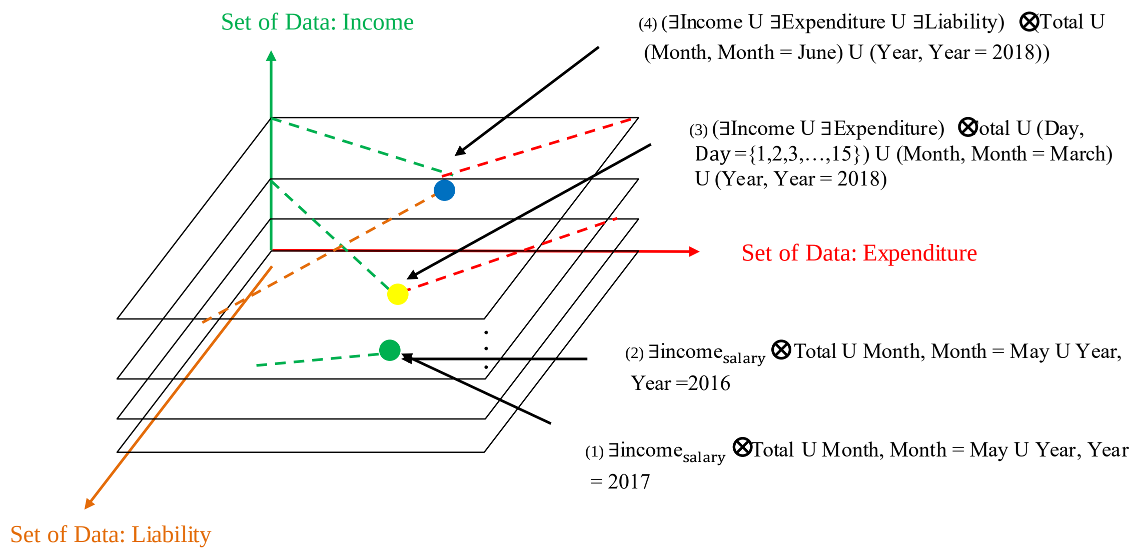
Set of Data: Income
(4) (∃Income U ∃Expenditure U ∃Liability) ⊗Total U (Month, Month = June) U (Year, Year = 2018))
(3) (∃Income U ∃Expenditure) ⊗otal U (Day, Day = {1,2,3,…,15}) U (Month, Month = March) U (Year, Year = 2018)
Set of Data: Expenditure
(2) ∃income$_{salary}$ ⊗Total U Month, Month = May U Year, Year = 2016
(1) ∃income$_{salary}$ ⊗Total U Month, Month = May U Year, Year = 2017
Set of Data: Liability

**Figure 12** Formulation of cutting points on three-dimensional layering

5.2 Circuit component analysis of the application

The house bookkeeping mobile application has been developed by C# .Net language on Xamarin cross-platform to deliver both iOS and Android applications. There are many functionalities of the house bookkeeping application. Each functionality has many components to make it worked. Using the RAD methodology to develop a prototype for users to test and verify functionalities, every prototype has to agree to accept the design and use aesthetic. We can move on to the next component to ensure that users are delighted and comfortable using the application. If users feel unsatisfied and uncomfortable to use the application, we must reconsider, redesign, implement, assess, and bring to users once more for their opinions. The house bookkeeping application consists of 16 primary components. Each component has subcomponents, as illustrated in Table 2. The first prototype included 156 subcomponents. Each change of the prototype will have to undergo many components with the same functionality and system properties in many components. When anyone changes the same functionality or a system property was missing from the other components, this situation inconsistent information of the component that may cause errors in the application. Using Information Flow Diagram (IFD) in the design and development, we have not confronted any missing information flows, processes, user interfaces, and databases. However, we have lost tracking component interactions, trading information, redundant functionalities, and system properties, and been tedious to make changes. Therefore, the second prototype took so many times to make changes and deploy the application to users.

We experienced using the concept of circuit components to analyze, reconsider, redesign, implement, correct, and cutover brings less time of development with the better comprehend, lesser tightly coupled, and higher cohesion. The multiconcerns circuit component diagram divides components into two types of components: aspectual components and functional data components. Each component visibly defines inputs, outputs, layers, and constraints. This technique leads the developers and designers to analyze and effortlessly refactor components improbably. We have found that the third prototype, using the information flow diagram with multiconcerns circuit component diagram, enables to design and develop the application with fewer components, interactions, information, processes, and the same functionalities and properties. In Table 2, we have achieved to reduce the duplicate components from 156 components to 113 components. Thus, the house bookkeeping mobile application's final prototype is one-fourth of the third prototype components, of 26.47% component reduction.

**Table 2** Comparison of components by using IFD without MCCD and IFD with MCCD

| Components | IFD without MCCD | IFD with MCCD | Component reduction | %Reduction |
|---|---|---|---|---|
| Welcome Screens | 10 | 10 | 0 | **0** |
| User Registering | 5 | 4 | 1 | **20** |
| User Authentication | 4 | 3 | 1 | **25** |
| Multilanguage | 9 | 9 | 0 | **0** |
| Income Transaction | 11 | 9 | 2 | **18.19** |
| Expenses Transactions | 11 | 9 | 2 | **18.19** |
| Liability Transaction | 11 | 9 | 2 | **18.19** |
| Menu Screens | 4 | 4 | 0 | **0** |
| Financial Reports | 17 | 14 | 3 | **17.65** |
| Balance Calculation | 6 | 1 | 5 | **83.33** |
| Account Management | 9 | 3 | 6 | **66.67** |
| Income Categories | 12 | 8 | 4 | **33.33** |
| Expenses Categories | 12 | 8 | 4 | **33.33** |
| Liability Categories | 12 | 8 | 4 | **33.33** |
| Calendar | 16 | 7 | 9 | **56.25** |
| User Profile | 7 | 7 | 7 | **0** |
| **Total** | **156** | **113** | **50** | **26.47** |

In Figure 13, we demonstrated the number of components in the house bookkeeping mobile application using an information flow diagram without a multiconcerns circuit diagram and the number of components in the ultimate house bookkeeping mobile application using an information flow multiconcerns circuit diagram.

The components of the house bookkeeping software design described in this paper have been proposed using a circuit's structures. Although the circuit is rarely applied to define a property of components, the software building block seems to contain input and an output that can be useful to interconnect several entities and describe supporting the separation of concerns. The advantage of input and output characteristics is to identify clearly between aspects and functional data during a weaving execution. We assign that inputs are composed of datasets, functional data, aspect elements, configurations, and outputs collected by crosscutting a method call. The various concerns and aspects are easy to use to represent ordinary relations of separating concerns in three-dimensional component layering. The illustration of Figure 13 presents the sample of cutting points that can show significantly different data even at the same point because of a plane layer. In this case, the Layer Interface Abstraction is designed to manipulate between layers of concerned circuits to ensure that the core recording of a house bookkeeping is divided relatively. However, a circuit component enables to prove the construction of components at a crosscutting point. A component working as an object is required weaving at joining points. Therefore, the syntax overview of explicit joint points should be declared for transforming the conceptual design of the multiconcerns circuit into programming.

For software design descriptions (SDDs) in this article, we use the IEEE standard 1016-2009 for Information Technology. The IEEE Std 1016-2009 stipulates that an SDD should be organized into a few design views. Each view addresses a specific set of design concerns of the stakeholders. Thus, we may state for the better approach of a design view, design entities, such as component, class, data stores, and process in advance to capture all critical elements for supporting design views. Moreover, the summary of design views has mentioned that the composition is refined into new viewpoints. We are genuinely sure that our components of a concerned circuit can be express physical designs and logical decompositions of functionality. Besides, the partitioning of information should be more increased by design viewpoints. To agree with this statement, in three-dimensional layering, a plane of layering has provided the hyperspace design of the sets of data and functional data. The house bookkeeping software enables support users to add their new sub-categories of recordings. For example, a user may have an extra job (selling products online), the user can add a new type of income. The report must include several new revenues by separating from each category.

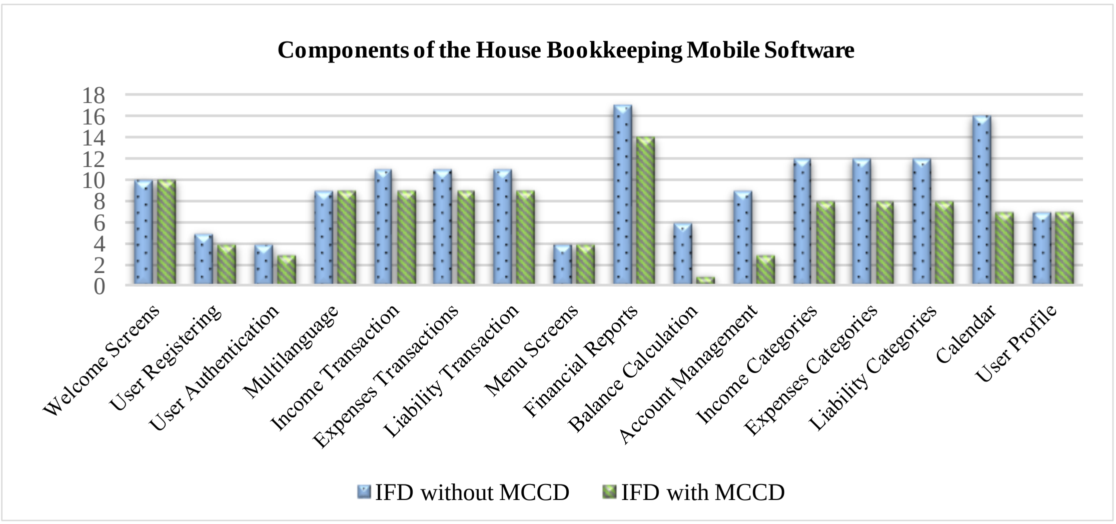

**Figure 13** Number of components in the application developed between IFD without MCCD and IFD with MCCD

The multiconcerns circuit component technique has shown that the component-based model concentrates on a fewer decomposition of tangling components while preserving exactly consistent former system operations. The multiconcerns circuit component technique provides interoperations of components into individual functional or aspectual components, precisely representing essential communications. However, other concerns of component-based software quality, such as system performance, reusability, extensibility, and adaptability, need to be supplemented to discover how this technique could accomplish it.

## 6. Discussion

In the early stages of approaching the separation of concerns, the user's requirement is gathered to analyze and design for non-functional requirements. Aspect-Oriented Architecture Design (AOAD) only improved non-functional requirements (Sanchez, Moreira, Fuentes, Araujo, & Magno, 2010; Panunzio~~,~~ & Vardanega, 2014). In our work, the separation of concerns is used to design the house bookkeeping software design's functional requirements. We have applied the separation of concerns by dividing a fine granularity into sets of data, functional data, and aspect elements. The data sets are related to the three-dimensional layering for providing a method call through a plane on each dimension or more named the functional data. The hyperspace design can represent a crosscutting point between the functional data and the aspect elements. By dividing the aspectual properties from the base components, the separation of concerns can lead us to the dependencies of the aspect elements claimed by Sanchez, Moreira, Fuentes, Araujo, and Magno (2010).

Additionally, if there is a change in a development period, such as an extension request of functional components and a new method, call to a pointcut. The new aspects are called statically at a compiling time, and a dynamic weaving is provide supporting on the requirement of applications during a running time (Diaz, Romero, Rubio, Soler, & Troya, 2005; Zhang~~,~~ & Rong, 2009; Rukhiran~~,~~ & Netinant, 2020a). The new requirement will not impact the design and implementation phases. Design views of the multiconcerns circuit component diagram are provided separately using crosscutting points to execute between the functional data and the aspect elements. Weaving ideas can be programmed supporting any language (Java, C#, C++, and Python). Basically, in Java, there are many programming extensions, including AspectJ (a widespread aspect-oriented extension to Java) and weaveJ (a dynamic aspect weaver for Java Virtual Machine (JVM) which uses a special opcode (invokedynamic method)) to be added in an implemented code (Garcia, Ortin, Llewellyn-Jones, & Merabti, 2013; Rodriguez-Prieto, Ortin, & O'Shea, 2018). AspectJ is a simple aspect-oriented extension supporting operating systems and embedded systems programming. In this paper, the house bookkeeping mobile application has been developed by C# .Net language on Xamarin cross-platform. The multiconcerns circuit component diagram establishes the separation of functional components and aspectual components. The diagram makes developers visualize, comprehend, realize, and assembled the components to reduce redundant components, have few tangling components, refactor unstructured components, and be better interactions of components.

By designing the house bookkeeping software components, the execution flow of the multiconcerns circuit component diagram can represent the individual work of each component, information, relationships, and interactions between components. While all components are proposed by describing information, processes, structures, interactions, and functions, we have found that the functional cooperation between components is an important key that should identify clearly for increasing modularity and ease of reuse components (Hoffman~~,~~ & Eugster, 2008). Many inputs and outputs signals of a circuit component can represent connections and interaction to perform different crosscutting concerns at a running time. Design views are applied using the concept of viewpoint. The viewpoints approach allocates a few points with another corresponding (Panunzio~~,~~ & Vardanega, 2014). The component-based software is proposed as a component model involving the improved separation of concerns and minimal interaction environments.

## 7. Conclusion

In this paper, the multiconcerns circuit component diagram promises to support a better aspect-oriented approach through a different glass of coffee to describe how the aspect-oriented approach works differently. The three-dimensional layering is proposed diving from three different categories of the house bookkeeping software design. The functional requirements are influenced to design the aspect elements of the house bookkeeping generated and categorized in two types for supporting calling once time and repeating at the same time. A multilayered approach is provided for adapting a variety of crosscutting concerns by weaving. Weaving is an operation between functional data and the aspect elements. For improving software designs, a developer should focus on designing an architecture constraint known as components. Designing components enable collaboration of the reusable and flexible classes. We have defined Layer Interface Abstraction as a conjunction layering between a component-based circuit component diagram and a low-level perspective of separating multiconcerns circuit components (an income layering, an expenditure layering, and a liability layering). The layer interface abstraction is more beneficial for independent concerns and data granularity. Thus, the evaluation presenting through operational semantics of three-dimensional layering enables understanding the executions of weaving and ease to use in an implementation phase. We have compared the house bookkeeping software design and development using between the information flow diagram without the multiconcerns circuit component diagram and the information flow diagram with the multiconcerns circuit component diagram. The component layering is not guaranteed to lead to a great deal of complexity and duplication in designing, developing, and maintaining software. Software engineers are responsible for dealing with those issues by using any techniques. Therefore, component-based software is enabled to apply by the new introduction of multiconcerns circuit components.

## 8. Acknowledgements

This work was supported by the National Broadcasting and Telecommunication Commission (grant number BT2-15/1-61) during 2019 - 2021, Rangsit University, and Rajamangala University of Technology Tawan-OK, Thailand. This research project was funded in the amount of $100,000USD.

## 9. References

AI-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

Benedetto, P. D. (2011). *A framework for context aware adaptable software services: A framework for programming, analyzing, delivering and deploying context-aware adaptable applications and services*. Saarbrucken, Germany: LAP LAMBERT Academic Publishing.

Boukraa, D., Boussaid, O., Bentayeb, F., & Zegour, D. (2013). A layered multidimensional model of complex objects. *Proceeding of the 25th International Conference on Advanced Information Systems Engineering*. 17-21 June, 2013. Valencia, Spain. pp. 498-513. Valencia, Spain: Springer-Verlag.

Chomngern, T., & Netinant, P. (2017). A mobile software model for web-based learning using information flow diagram. *Proceeding of the ACM International conference on information technology*. December 27-29, 2017. Singapore, Singapore. pp. 243-247. DOI: 10.1145/3176653.3176680

Diaz, M., Romero, S., Rubio, B., Soler, E., & Troya, J. M. (2005). An aspect oriented framework for scientific component development. *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing*. February 9-12, 2005. Washington, USA. pp. 290-296. DOI: 10.1109/EMPDP.2005.11

Gama, K. & Donsez, D. (2011). Applying dependability aspects on top of aspectized software layers. *Proceedings of the 10th international conference on Aspect-oriented software development*. March 21-25, 2011. Pernambuco, Brazil. pp. 177-190. DOI: 10.1145/1960275.1960297

Garcia, M., Ortin, F., Llewellyn-Jones, D., & Merabti, M. A. (2013). Performance cost evaluation of aspect weaving. *Proceedings of the 36th Australasian Computer Science Conference*. January 29-February 1, 2013. Adelaide, Australia. pp. 79-85. DOI: 10.5555/2525401.2525410

Giedrimas, V., & Omanovic, S. (2015). The impact of mobile architectures on component-based software engineering. *Proceeding of the 3rd Workshop on Advances in Information, Electronic and Electrical Engineering*. November 13-14, 2015. Riga, Latvia. pp. 1- 6. DOI: 10.1109/AIEEE.2015.7367317

Gupta K. S., Singh, J., & Kumar, M. (2016). Composing an aspect oriented approach to synchronization problems. *Proceeding of the 3rd International Conference on Computing for Sustainable Global Development*. March

16-18, 2016. New Delhi, India. pp. 3036-3041

Hoffman, K., & Eugster, P. (2008). Towards reusable components with aspects: An empirical study on modularity and obliviousness. *Proceedings of the 30th International Conference on Software Engineering*. May 10-18, 2008. Leipzig, Germany. pp. 91-100. DOI: 10.1145/1368088.1368102

Kebir, S. (2012). JACAC : An aspect oriented framework for the development of self-adaptive software systems. *Proceeding of the 6th International Conference on Sciences of Electronics, Technologies of Information and Telecommunications*. March 21-24, 2012. Sousse, Tunisia. pp. 74-80. DOI: 10.1109/SETIT.2012.6481893

Khanzadi, M., Shahbazi1, 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*, 1307-1316. DOI: 10.1007/s40999-018-00389-9

Kumar, A., Kumar, A., & Iyyappan, M. (2016). Applying separation of concern for developing softwares using aspect oriented programming concepts. *Procedia Computer Science, 8*, 906-914. DOI: 10.1016/j.procs.2016.05.281

Lee, J., & Bae, D. (2004). An aspect-oriented framework for developing component-based software with the collaboration-based architectural style. *Information and Software Technology, 46*(2), 81-97. DOI: 10.1016/S0950-5849(03)00111-3

Lindstrom, B., Offutt, J., Sundmark, D., Andler, 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. 04.007

Lobato, C., Garcia, A., Romanovsky, A., & Lucena, C. (2008). An aspect-oriented software architecture for code mobility. *Software-Practice & Experience, 38*(13), 1365-1392. DOI: 10.5555/1455460.1455462

Muck, T. R., & Frohlich, A. A. (2014). Aspect-oriented RTL HW design using system C. *Microprocessors and Microsystems, 38*, 113-123. DOI: 10.1016/j.micpro.2013.12.002

Netinant, P., & Elrad, T. (2016). Separation of concerns in designing mobile software. *Journal of Current Science and Technology, 6*(1), 89-96. DOI: 10.14456/rjas.2016.8

Okewu, E., & Daramola, O. (2014). Component-based software engineering approach to development of a university e-Administration System. *Proceeding of the 6th International Conference on Adaptive Science & Technology*. Ota, Nigeria. October 29-31, 2014. pp. 1-8. DOI: 10.1109/ICASTECH.2014.7068152

Panunzio, M., & Vardanega, T. (2014). A component-based process with separation of concerns for the development of embedded real-time software systems. *The Journal of Systems and Software, 96*, 105-121. DOI: 10.1016/j.jss.2014.05.076

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, in press*, 1-14. DOI: 10.1016/j.jksuci.2020.10.029

Rodriguez-Prieto, O., Ortin, F. & O'Shea, D. (2018). Efficient runtime aspect weaving for Java applications. *Journal of Information and Software Technology, 100*, 73-86. DOI: 10.1016/j.infsof.2018.03.012

Rukhiran, M, & Netinant, P. (2020a). A practical model from multidimensional layering personal financial information framework to mobile software interface operations. *Journal of Information and Communication Technology, 19*(3), 321-349.

Rukhiran, M., & Netinant, P. (2020b). IoT architecture based on information flow diagram for vermiculture smart farming kit. *TEM Journal, 9*(4), 1330-1337. DOI: 10.18421/TEM94-03

Sadeghi, A., Esfahani, N., & Malek, S. (2017). Ensuring the consistency of adaptation through inter- and intra-component dependency analysis. *ACM Transactions on Software Engineering and Methodology, 26*(2), 2:1-2:27. DOI: 10.1145/3063385

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. DOI:10.1016/j.infsof.2009.09.001

Sommerville, I. (2014). *Software Engineering* (10th ed.). Boston, Massachusetts: Pearson Education, Inc.

Tanter, E., Figueroa, I., & Tabaerau, N. (2014). Execution levels for aspect-oriented programming: Design, semantics, implementations and applications. *Science of Computer Programming, 80,* 311-342. DOI: 10.1016/j.scico.2013.09.002

Tibermacine, C., Sadou, S., Dony, C., & Fabresse, L. (2011). Component-based specification of software architecture constraints. *Proceedings of the 14th international ACM Sigsoft symposium on Component based software engineering*. June 21-23, 2011. New York, United States. pp 31-40. DOI: 10.1145/2000229.2000235

Verma, I. (2002). Component-based software engineering. *International Journal of Computer Science & Communication Networks, 4*(3), 84-88.

Witana, V., Fry, M., & Antoniades, M. (1999). A software framework for application-level QoS management. *Proceeding of the 7th International Workshop on Quality of Service*. May 31-June 4, 1999. London, United Kingdom. pp: 51-61. DOI: 10.1109/IWQOS.1999.766478

Zhang, G., & Rong, M. (2009). A framework for dynamic evolution based on reflective aspect-oriented software architecture. *Proceedings of the 4th International Conference on Computer Sciences and Convergence Information Technology*. November 24- 26, 2009. Seoul, South Korea. pp. 7-10. DOI: 10.1109/ICCIT.2009.102

Zhang, Q., Khedri, R., & Jaskolka, J. (2012). An aspect-oriented language for product family specification. *Procedia Computer Science, 10*, 482-489. DOI: 10.1016/j.procs.2012.06.062