Aspect Oriented Software Development vs. other Techniques (Structured Approach and Object Oriented Approach)

Ahmed Yakout A. Mohamed
Arab Academy for Science, Technology & Maritime Transport
College of Computing and Information Technology
Tel: 2-011-8606-888 E-mail: systems.analyst@hotmail.com

Prof Dr. Abd El Fatah .A. Hegazy
Arab Academy for Science, Technology & Maritime Transport
College of Computing and Information Technology
E-mail: ahegazy@aast.edu

Dr. Ahmed R.Dawood
Chairman, MIS Department
Arab Academy for Science, Technology & Maritime Transport
E-mail: ahmedredadawoad@yahoo.com

Abstract

Aspects are a natural evolution of the object-oriented paradigm. They provide a solution to some difficulties you may have encountered with modularizing your object-oriented code: sometimes functionality just doesn't fit! You've probably found yourself repeating the same lines of code in lots of different object-oriented classes because those classes each need that functionality, and so you can't easily wrap it up in a single place. Good examples of this kind of code are audit trails, transaction handling, concurrency management, and so on. You can now modularize such code with aspects. Aspect-Oriented Software Development (AOSD). Provides unique and advanced program structuring and modularization techniques. The implementation of software applications using AOSD techniques results in a better implementation structure which has an impact on many important software qualities such as enhanced reusability and reduced complexity. In turn, these software qualities lead to an improved software development lifecycle and, hence, to better software.

Keywords: Aspect – Orientation, Traditional engineering, Object – Oriented

1. Introduction

Aspect-Orientation (AO) is in an exciting phase of growth, but that means that new languages and new possibilities are coming out frequently, and that the basic notions of an "aspect" shifts subtly as new philosophies are revealed. There are different styles of decomposition, even within aspect-orientation (Siobhán C, & Elisa B 2005).

Traditional engineering disciplines manage the complexity of systems by identifying and separating the system's concerns and treating each concern in isolation; such an approach known as separation of concerns (SOC), leads to systems that are easier to implement, verify, evolve, and understand. It is desirable to decompose a system into modules in such a way that modules responsible for a common concern are tightly coupled and modules responsible for different concerns are loosely coupled. AOSD is a post object-oriented technology that helps achieve better SOC by providing mechanisms to localize crosscutting concerns or aspects (e.g. security, logging, etc.) in software artifacts throughout the software development process (Shaker P. , & Peters D 2005).

AOSD techniques provide a systematic means for identification, modularization, representation, and composition of crosscutting concerns (A. Rashid, A. Moreira, & J. Araujo 2003). The term crosscutting concerns (e.g., reliability, synchronization) refers to such quality factors or functionalities of software that cannot be effectively modularized using existing software development techniques, e.g., object-oriented (OO) approaches.

AOSD techniques build on existing work in software development techniques and methodologies in order to tackle such concerns in a systematic fashion. Though most of the initial work in AOSD has focused on developing of aspect-oriented programming (AOP) languages, frameworks and platforms, a number of methods and techniques have also focused on addressing Crosscutting concerns at the analysis and design level. Consequently, a significant body of research exists in the area of Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design (A. Rashid 2005) as well as in modeling and design of systems derived from such aspectual requirements and architectures. The goal of this report is to undertake a survey of the state-of-the-art in handling crosscutting concerns, both in contemporary analysis and design approaches as well as those based on AOSD principles and practice.

As a result, there arouse benefits of applying the Aspect oriented methodology to earlier stages of the software development cycle as it ensures improved modularity at all stages of the development process where preserving the notion of aspects throughout the development process ensures traceability (AOSD 2009).

A method to software development based around aspects is mentioned as Aspect oriented methodology.

This software development methodology focuses on:

- Identifying cross-cutting concerns.
- Reduces system complexity through separation of concerns.
- Separation of concerns is to represent concern independently (Georg G., & Ray 2009).

Aspects provide active support, not just textual code manipulation, for separating concerns in source code. Aspects have been applied to far more complex crosscutting concerns than synchronization, logging, and tracing. For example, they have been applied in operating systems as a way to encapsulate and improve their performance (Yvonne Coady, & Gregor Kiczales 2003).

2. The Traditional (Structured) Approach

- Traditional Approach includes many variations based on techniques used to develop information system with structured and modular programming.
- Traditional Approach also known as Structured System Development.
- Traditional Approach is one particular implementation and builds on the work of different schools of “structured analysis” and development methods, such as Peter Checkland's “Soft Systems Methodology”, Larry Constantine's “Structured Design”, Edward 'Yourdon's “Yourdon Structured Method”, Michael A. Jackson's “Jackson Structured Programming”, and Tom DeMarco's “Structured Analysis”(SSADM 2009).
- The names "Structured Systems Analysis and Design Method" and "SSADM" are now “Registered Trade Marks” of the “Office of Government Commerce” (OGC), which is an Office of the United Kingdom's Treasury.

2.1 Structured analysis and design technique (SADT)

- Structured approach which as known as Structured analysis and design technique (SADT).
- SADT is a software engineering methodology for describing systems as a hierarchy of functions.
- SADT is a graphical language that for describing systems, together with a methodology for producing such as description.
- SADT basis elements: is shown in Figure 1.
- Diagrammatic notation designed specifically to help people describe and understand systems.
- Offers building blocks (boxes) to represent entities and activities.
- Offers variety of arrows to relate boxes.
- Boxes and arrows have an associated (informal) semantics; users are aided by boxes and arrow labels, other informal documentation. (Levitt, D 2000).
- There are 3 techniques in SADT :

2.1.1. Structured analysis (SA).

- A technique use to define what processing the system needs to do, what data it needs to store and used, and what inputs and outputs are needed (External approach : events and reactions to event) (Internal approach : functions , inputs , outputs)

- In structured analysis there are three orthogonal views:
  - The functional view, made up of data flow diagrams, is the primary view of the system. It defines what is done, the flow of data between things that are done and provides the primary structure of the solution.
  - The data view, made up of entity relationship diagrams, is a record of what is in the system, or what is outside the system that is being monitored. It is the static structural view.
  - The dynamic view, made up of state transition diagrams, defines when things happen and the conditions under which they happen.
- The result of structured analysis is a set of related graphical diagrams, process descriptions, and data definitions.
- Approach for structured analysis consists of the following objects :
  - Data Flow Diagrams (DFD) is shown in Figure 2

1) Shows processes and flow of data in and out of these processes.

2) Does not show control structures (loops)?

3) Contains 5 graphic symbols (shown later).

4) Uses layers to decompose complex systems (show later).

5) Can be used to show logical and physical.

6) Were quantum leaps forward to other techniques at the time, I.e. monolithic descriptions with globs of text (Levitt, D 2000)?

  - Context Diagram is shown in Figure 3

1) Represent all external entities that may interact with a system.

2) This diagram pictures the system at the center, with no details of its interior structure, surrounding by all its interacting systems, environment and activities.

3) The objective of a system context diagram is to focus attention on external factors and events that should be considered in developing a complete set of system requirements and constrains.

4) System context diagram are related to Data Flow Diagram.

5) Context diagrams can be helpful in understanding the context in which the system will be part of software engineering.

  - Data Dictionary is shown in Figure 4

1) A data dictionary or database dictionary is a file that defines the basic organization of a database.

2) A database dictionary contains a list of all files in the database, the number of records in each file, and the names and types of each data field.

3) Most database management systems keep the data dictionary hidden from users to prevent them from accidentally destroying its contents.

4) Data dictionaries do not contain any actual data from the database, only book keeping information for managing it.

2.1.2. Structured design (SD): is shown in Figure 5.

- Methods for analyzing and converting business requirements into specifications and ultimately, computer programs, hardware configurations and related manual procedures
- A techniques providing guidelines for deciding what the set of programs should be , what each program should accomplish and how the programs should be organized into a hierarchy
- Structured Design views the word as a collection of modules with functions that share data with other (sub) modules. Example: structure chart
- Structured Design addresses the synthesis of a module hierarchy.
- The principles of cohesion and coupling are applied to derive an optimal module structure and interfaces.
- Cohesion is concerned with the grouping of functionally related processes into a particular module.

- Coupling addresses the flow of information, or parameters, passed between modules. Optimal coupling reduces the interfaces of modules, and the resulting complexity of the software.
- The structure chart shows the module hierarchy or calling sequence relationship of modules. There is a module specification for each module shown on the structure chart.
- The module specifications can be composed of pseudo-code or a program design language. The data dictionary is like that of structured analysis. At this stage in the software development lifecycle, after analysis and design have been performed, it is possible to automatically generate data type declarations, and procedure or subroutine templates. (Mohammad. R 2004)

2.1.2.1. Logical data design

Also known as: logical system specification stage. In this stage, technically feasible options are chosen. The development/implementation environments are specified based on this choice. The following steps are part of this stage:

- Define TSOs: Up to 6 technical options (specifying the development and implementation environments) is produced, one being selected.
- Select TSOs. Select the most favorable TSO.

2.1.2.2. Logical process design

Also known as: logical system specification stage. In this stage, logical designs and processes are updated. Additionally, the dialogs are specified as well. The following steps are part of this stage:

- Define user dialogue. This step defines the structure of each dialogue required to support the on-line functions and identifies the navigation requirements, both within the dialogue and between dialogues.
- Define update processes. This is to complete the specification of the database updating required for each event and to define the error handling for each event.
- Define inquiry processes. This is to complete the specification of the database enquiry processing and to define the error handling for each inquiry.

2.1.2.3 Physical design

The objective of this stage is to specify the physical data and process design, using the language and features of the chosen physical environment and incorporating installation standards. The following activities are part of this stage:

- Prepare for physical design
- Learn the rules of the implementation environment
- Review the precise requirements for logical to physical mapping
- Plan the approach
- Complete the specification of functions
- Incrementally and repeatedly develop the data and process designs(Mylopoulos J 2002)

2.2 Decomposition Structure: is shown in Figure 6

The structured analysis and design technique uses decomposition with the top-down approach. This decomposition is conducted only in the physical domain from an axiomatic design viewpoint. Because of this non-zigzagging process, there is no guarantee of functionality or productivity. Therefore, those methods faded away as the requirements for software systems increased and the object-oriented method was introduced (Nam Pyo Suh 2007).

3. Object – Oriented Approach

3.1 Introduction

- An approach to system development that views information system as a collection of interacting objects that work together to accomplish task
- The OO approach is a different way of developing software.
- Object is a things in computer system that can respond to messages
- Object oriented notions :

- Class: metadata, i.e. definition of data structures, methods by which they can be operated and behavior of class entities (i.e. interface with entities)
- Object: instances of class, i.e. contains an actual data, code, input and output;
- Object has observable behavior: can process data, send and respond to messages
- Information system is collection of interacting objects that work together to accomplish tasks
- Main concepts: instantiation and inheritance

- Conceptually , no process , programs , data entities or files are defined – just objects (OOAD 2009)

3.2 Object-oriented systems

An object-oriented system is composed of objects. The behavior of the system results from the collaboration of those objects. Collaboration between objects involves those sending messages to each other. Sending a message differs from calling a function in that when a target object receives a message, it itself decides what function to carry out to service that message. (OOAD 2009)

3.3 Object-oriented analysis (OOA)

- Defining all the types of objects that do the work in the system and showing what use cases are required to complete tasks
- OOA applies object-modeling techniques to analyze the functional requirements for a system
- Looks at the problem domain, with the aim of producing a conceptual model of the information that exists in the area being analyzed.
- Analysis models do not consider any implementation constraints that might exist, such as concurrency, distribution, persistence, or how the system is to be built.
- The sources for the analysis can be a written requirements statement, a formal vision document, and interviews with stakeholders or other interested parties.
- A system may be divided into multiple domains, representing different business, technological, or other areas of interest, each of which are analyzed separately.
- The result of object-oriented analysis is a description of what the system is functionally required to do, in the form of a conceptual model. (TDD 2009)
- That will typically be presented :
  - A set of use cases; A use case in software engineering and systems engineering is a description of a system's behavior as it responds to a request that originates from outside of that system. In other words, a use case describes "who" can do "what" with the system in question. The use case technique is used to capture a system's behavioral requirements by detailing scenario-driven threads through the functional requirements.
  - One or more UML class diagrams; Unified Modeling Language (UML) is a standardized general-purpose modeling language in the field of software engineering. UML includes a set of graphical notation techniques to create visual models of software-intensive systems is shown in Figure 7.
  - A number of interaction diagrams; Interaction diagrams, a subset of behavior diagrams, emphasize the flow of control and data among the things in the system being modeled:

1) Communication diagram: shows the interactions between objects or parts in terms of sequenced messages. They represent a combination of information taken from Class, Sequence, and Use Case Diagrams describing both the static structure and dynamic behavior of a system is shown in Figure 8.

2) Interaction overview diagram: are types of activity diagram in which the nodes represent interaction diagrams is shown in Figure 9.

3) Sequence diagram: shows how objects communicate with each other in terms of a sequence of messages. Also indicates the lifespan of objects relative to those messages is shown in Figure 10.

4) Timing diagrams: are specific types of interaction diagram, where the focus is on timing constraints is shown in Figure 11.

- It may also include some kind of user interface mock-up. object oriented analysis is to develops a model that describe computer software as it works to satisfy a set of customer define requirements'

3.4 Object-oriented design (OOD)

- Defining all of the types of object necessary to communicate with people and devices in the system , showing how objects interact to complete tasks and refining the definitions of each type of object so it can be implemented with specific language or environment
- Object-oriented design is the process of planning a system of interacting objects for the purpose of solving a software problem.
- The input for object-oriented design is provided by the output of object-oriented analysis.
- Realize that an output artifact does not need to be completely developed to serve as input of object-oriented design; analysis and design may occur in parallel, and in practice the results of one activity can feed the other in a short feedback cycle through an iterative process.
- Both analysis and design can be performed incrementally, and the artifacts can be continuously grown instead of completely developed in one shot.
- Some typical input artifacts for object-oriented design are:
  - Conceptual model
  - Use case
  - System Sequence Diagram.
  - User interface documentations
  - Relational data model
- Class diagrams derived from conceptual diagram and use cases
- Class diagrams a graphical model used in the object-oriented approach to show classes of objects in the system
- Elements of class diagrams are:
  - Classes, metadata describing association of data structures with the methods or functions that act on the data.
  - Links describe relationship between classes.
  - Types of links: Association, aggregation, composition.
  - Classes relate in class relationship that can be inheritance ,dependency, sub-type ,super-type
- Class diagrams derived from conceptual diagram and use cases (W. John, Jackson. R, & D. Bur. Stephen 2005) is shown in Figure 12.

3.5 Object-Oriented Modeling

Object-Oriented Modeling, or OOM, is a modeling paradigm mainly used in computer programming. Prior to the rise of OOM, the dominant paradigm was procedural programming, which emphasized the use of discreet reusable code blocks that could stand on their own, take variables, perform a function on them, and return values.

The Object-Oriented paradigm assists the programmer to address the complexity of a problem domain by considering the problem not as a set of functions that can be performed but primarily as a set of related, interacting Objects. The modeling task then is specifying, for a specific context, those Objects (or the Class the Objects belongs to), their respective set of Properties and Methods, shared by all Objects members of the Class (OOM 2009).

3.5.1 Software Development Methods

UML is not a development method by itself,(John Hunt 2000) however, it was designed to be compatible with the leading object-oriented software development methods of its time (for example OMT, Booch method, Object). Since UML has evolved, some of these methods have been recast to take advantage of the new notations (for example OMT), and new methods have been created based on UML. The best known is IBM Rational Unified Process (RUP).

3.5.2 Modeling

It is very important to distinguish between the UML model and the set of diagrams of a system. A diagram is a partial graphical representation of a system's model. The model also contains "semantic backplane" documentation such as written use cases that drive the model elements and diagrams.

UML diagrams represent two different views of a system model (Jon Holt 2004):

- Static (or structural) view: Emphasizes the static structure of the system using objects, attributes, operations and relationships. The structural view includes class diagrams and composite structure diagrams.
- Dynamic (or behavioral) view: Emphasizes the dynamic behavior of the system by showing collaborations among objects and changes to the internal states of objects. This view includes sequence diagrams, activity diagrams and state machine diagrams.

UML models can be exchanged among UML tools by using the XMI interchange format.

3.5.3 Diagrams overview

UML 2.2 has 14 types of diagrams divided into two categories.(OMG 2009) Seven diagram types represent structural information, and the other seven represent general types of behavior, including four that represent different aspects of interactions. These diagrams can be categorized hierarchically as shown in the following class diagram is shown in Figure 13.

UML does not restrict UML element types to a certain diagram type. In general, every UML element may appear on almost all types of diagrams; this flexibility has been partially restricted in UML 2.0. UML profiles may define additional diagram types or extend existing diagrams with additional notations.

In keeping with the tradition of engineering drawings, a comment or note explaining usage, constraint, or intent is allowed in a UML diagram.

3.5.3.1 Structure diagrams

Structure diagrams emphasize what things must be in the system being modeled:

- Class diagram: The class diagram describes the structure of a system by showing the system's classes, their attributes, and the relationships among the classes as showing in Figure 14.
- Component diagram: depicts how a software system is split up into components and shows the dependencies among these components as showing in Figure 15.
- Composite structure diagram: describes the internal structure of a class and the collaborations that this structure makes possible as showing in Figure 16.
- Deployment diagram: serves to model the hardware used in system implementations, and the execution environments and artifacts deployed on the hardware as showing in Figure 17.
- Object diagram: shows a complete or partial view of the structure of a modeled system at a specific time as showing in Figure 18.
- Package diagram: depicts how a system is split up into logical groupings by showing the dependencies among these groupings as showing in Figure 19.
- Profile diagram: operates at the meta model level to show stereotypes as classes with the <<stereotype>> stereotype, and profiles as packages with the <<profile>> stereotype. The extension relation (solid line with closed, filled arrowhead) indicates what meta model element a given stereotype is extending.

Since structure diagrams represent the structure they are used extensively in documenting the architecture of software systems.

3.5.3.2 Behavior diagrams

Behavior diagrams emphasize what must happen in the system being modeled:

- Activity diagram: represents the business and operational step-by-step workflows of components in a system. An activity diagram shows the overall flow of control as showing in Figure 20.
- State machine diagram: standardized notation to describe many systems, from computer programs to business processes as showing in Figure 21.
- Use case diagram: shows the functionality provided by a system in terms of actors, their goals represented as use cases, and any dependencies among those use cases as showing in Figure 22.

Since behavior diagrams illustrate the behavior of a system, they are used extensively to describe the functionality of software systems.

3.5.3.2.1 Interaction diagrams

Interaction diagrams, a subset of behavior diagrams, emphasize the flow of control and data among the things in the system being modeled:

- Communication diagram: shows the interactions between objects or parts in terms of sequenced messages. They represent a combination of information taken from Class, Sequence, and Use Case Diagrams describing both the static structure and dynamic behavior of a system as showing in Figure 23.
- Interaction overview diagram: are types of activity diagram in which the nodes represent interaction diagrams as showing in Figure 24.
- Sequence diagram: shows how objects communicate with each other in terms of a sequence of messages. Also indicates the life spans of objects relative to those messages as showing in Figure 25.
- Timing diagrams: are specific types of interaction diagram, where the focus is on timing constraints as showing in Figure 11.

The Protocol State Machine is a sub-variant of the State Machine. It may be used to model network communication protocols.

4. Aspect-oriented software development

Aspect-oriented software development (AOSD) is an emerging software development technology that seeks new modularizations of software systems in order to isolate secondary or supporting functions from the main program's business logic. AOSD allows multiple concerns to be expressed separately and automatically unified into working systems.

Traditional software development has focused on decomposing systems into units of primary functionality, while recognizing that there are other issues of concern that do not fit well into the primary decomposition. The traditional development process leaves it to the programmers to code modules corresponding to the primary functionality and to make sure that all other issues of concern are addressed in the code wherever appropriate. Programmers need to keep in mind all the things that need to be done, how to deal with each issue, the problems associated with the possible interactions, and the execution of the right behavior at the right time. These concerns span multiple primary functional units within the application, and often result in serious problems faced during application development and maintenance. The distribution of the code for realizing a concern becomes especially critical as the requirements for that concern evolve a system maintainer must find and correctly update a variety of situations.

AOSD focuses on the identification, specification and representation of crosscutting concerns and their modularization into separate functional units as well as their automated composition into a working system.

4.1 Crosscutting concerns

The motivation for aspect-oriented programming approaches stem from the problems caused by code scattering and tangling. The purpose of Aspect-Oriented Software Development is to provide systematic means to modularize crosscutting concerns.

The implementation of a concern is scattered if its code is spread out over multiple modules. The concern affects the implementation of multiple modules. Its implementation is not modular.

The implementation of a concern is tangled if its code is intermixed with code that implements other concerns. The module in which tangling occurs is not cohesive.

Scattering and tangling often go together, even though they are different concepts.

Aspect-oriented software development considers that code scattering and tangling are the symptoms of crosscutting concerns. Crosscutting concerns can not be modularized using the decomposition mechanisms of the language (object or procedures) because they inherently follow different decomposition rules. The implementation and integration of these concerns with the primary functional decomposition of the system causes code tangling and scattering.

4.1.1 Examples of crosscutting concerns

Examples of concerns that tend to be crosscutting include:

- Real-time constraints

- Error detection and correction
- Caching
- Logging

4.1.2 Problems caused by scattering and tangling

Scattering and tangling of behavior are the symptoms that the implementation of a concern is not well modularized. A concern that is not modularized does not exhibit a well defined interface. The interactions between the implementation of the concern and the modules of the system are not explicitly declared. They are encoded implicitly through the dependencies and interactions between fragments of code that implement the concern and the implementation of other modules.

4.2 System development

A module is primarily a unit of independent development. It can be implemented to a large extent independently of other modules. Modularity is achieved through the definition of well defined interfaces between segments of the system.(Parnas, D.L 1972)

The lack of explicit interfaces between crosscutting concerns and the modules obtained through the functional decomposition of the system imply that the implementation of these concerns, as well as the responsibility with respect to the correct implementation of these concerns, cannot be assigned to independent development teams. This responsibility has to be shared among different developers that work on the implementation of different modules of the system and have to integrate the crosscutting concern with the module behavior.

Furthermore, modules whose implementation is tangled with crosscutting concerns are hard to reuse in different contexts. Crosscutting impedes reuse of components. The lack of interfaces between crosscutting concerns and other modules makes it hard to represent and reason about the overall architecture of a system. As the concern is not modularized, the interactions between the concern and the top-level components of the system are hard to represent explicitly. Hence, these concerns become hard to reason about because the dependencies between crosscutting concerns and components are not specified.

4.3 Nature of aspect-orientation

The focus of Aspect-Oriented Software Development (AOSD) is in the investigation and implementation of new structures for software modularity that provide support for explicit abstractions to modularize concerns. Aspect-Oriented Programming approaches provide explicit abstractions for the modular implementation of concerns in design, code, documentation, or other artifacts developed during the software life-cycle. These modularized concerns are called aspects, and aspect-oriented approaches provide methods to compose them. Some approaches denote a root concern as the base. Various approaches provide different flexibility with respect to composition of aspects.

4.4 Quantification and obliviousness

The best known definition of the nature of AOSD is due to Filman and Friedman, which characterized AOSD using the equation aspect orientation = quantification + obliviousness.

AOP can be understood as the desires to make quantified statements about the behavior of programs and to have these quantifications hold over programs written by oblivious programmers.

AOP is the desire to make statements of the form: In program P, whenever condition C arises, perform action A over a conventionally coded program P.

Obliviousness implies that a program has no knowledge of which aspects modify it where or when, whereas quantification refers to the ability of aspects to affect multiple points in the program (a b c Filman, R., & D. Friedman 2000).

4.5 Aspect-oriented requirement engineering

Aspect-oriented requirement Engineering (also referred to as "Early Aspects") focuses on the identification, specification and representation of crosscutting properties at the requirement level. Examples of such properties include security, mobility, availability and real-time constraints. Crosscutting properties are requirements, use cases or features that have a broadly-scoped effect on other requirements or architecture components.

Aspect-oriented requirements engineering approaches are techniques that explicitly recognize the importance of clearly addressing both functional and non-functional crosscutting concerns in addition to non-crosscutting ones. Therefore, these approaches focus on systematically and modularly treating, reasoning about, composing and

subsequently tracing crosscutting functional and non-functional concerns via suitable abstraction, representation and composition mechanisms tailored to the requirements engineering domain.

Specific areas of excellence under the denominator of AO requirements analysis are**:**

- The aspect-oriented requirements process itself.
- The aspect-oriented requirements notations.
- Aspect-oriented requirements tool support.
- Adoption and integration of aspect-oriented requirements engineering, and assessment/evaluation of aspect-oriented requirements.

4.6 Aspect-oriented system architecture

Aspect-oriented system architecture focuses on the localization and specification of crosscutting concerns in architectural designs. Crosscutting concerns that appear at the architectural level cannot be modularized by redefining the software architecture using conventional architectural abstractions. Aspect-oriented system architecture languages propose explicit mechanisms to identify, specify and evaluate aspects at the architecture design level.

Aspect-oriented architecture starts from the observation that we need to identify, specify and evaluate aspects explicitly at the architecture design level. Aspectual architecture approaches describe steps for identifying architectural aspects. This information is used to redesign a given architecture in which the architectural aspects are made explicit. In this regard, specific areas of excellence are:

- the aspect-oriented architecture process itself,
- the aspect-oriented architecture notations,
- Aspect-oriented architecture tool support,
- Adoption and integration of aspect-oriented architecture, and assessment/evaluation of aspect-oriented architecture.

4.7 Aspect-oriented modeling and design

Aspect-oriented design has the same objectives as any software design activity, i.e. characterizing and specifying the behavior and structure of the software system. Its unique contribution to software design lies in the fact that concerns that are necessarily scattered and tangled in more traditional approaches can be modularized. Typically, such an approach includes both a process and a language. The process takes as input requirements and produces a design model. The produced design model represents separate concerns and their relationships. The language provides constructs that can describe the elements to be represented in the design and the relationships that can exist between those elements. In particular, constructs are provided to support concern modularization and the specification of concern composition, with consideration for conflicts. Beyond that, the design of each individual modularized concern compares to standard software design.

Here, specific areas of excellence areas are:

- The aspect-oriented design process itself.
- The aspect-oriented design notations.
- Aspect-oriented design tool support.
- Adoption and integration of aspect-oriented design.
- Assessment/evaluation of aspect-oriented design.

4.8 Aspect-oriented middleware

Middleware and AOSD strongly complement each other. In general, areas of excellence consist of

- support for the application developer, which includes
  - The crucial concepts of aspect supporting middleware.
  - Aspect-oriented software development using a specific middleware, involving the aspect programming model, aspect deployment model, platform infrastructure, and services of the middleware.
- support for the middleware developer with respect to
  - Host-infrastructure middleware.

- Distribution middleware.
- Common middleware services.
- Domain-specific middleware services.

4.9 Case study for AOSD and SOC

Banking System Accounts transfer, An automated banking system for receiving payment from an authorized user and for transfer of funds to an authorized transferee's account at a bank in a banking network, and we need to:

- Check user authentications.
- Recodes add operations in the system logging.
- Split the transactions in another concern.

4.9.1 Problem before SOC: as showing in Figure 26.

4.9.2 Problem after SOC: as showing in Figure 27.

4.9.3 Transfer Concerns: as showing in Figure 28.

4.9.4 Banking System with SOC: as showing in Figure 29.

- Code is scattered when one concern (like logging) is spread over a number of modules (e.g., Transfer method).
- Modules end up tangled with multiple concerns (e.g., Transfer, logging, and Authentication)

4.9.5 Aspect Solution

- AOSD attempts to solve this problem by allowing the programmer to express cross-cutting concerns (e.g., Transaction, logging, and Authentication) in stand-alone modules called aspects.
- Used in conjunction with other approaches - normally object-oriented software engineering.
- Aspects encapsulate functionality that cross-cuts and co-exists with other functionality.
- Aspects include a definition of where they should be included in a program as well as code implementing the cross-cutting concern.

4.9.6 Aspect Structure: as showing in Figure 30 example of authentication code.

5. Conclusions

AOSD is not the perfect solution to the issues involved in software production, but it is a good starting point if you haven't formulated your own yet. With experience, you should be able to fine tune the concepts of AOSD to better fit your own work environment. Personally, I believe that language extensions are completely the wrong direction to take. If anything, they defeat the purpose of AOSD because, at least with AspectJ, it seems that even tighter coupling between components results. Furthermore, a separate specification of program flow merely adds to the programmer's confusion and need to keep track of separate but related implementation details. Instead, the issues that AOSD raises can be better addressed within the existing tools (such as design patterns) and languages (such as C#, C++) by designing appropriate frameworks to manage crosscutting (Marc lives 2010).

Aspect-oriented software development is a relatively new and increasingly promising approach to software development. It offers a number of potential benefits for software engineering, mainly deriving from advanced approaches to separation of concerns, including (multidimensional) concern modeling, encapsulation, extraction, and composition. These may enable the development and evolution of software on a higher (concern-oriented) semantic level with unprecedented control and flexibility.

We see that the central problem of aspect technologies, is not just about crosscutting or SOC, but it involves deeper research about how to understand a number of software parts as separated artifacts and then integrate some of them into a coherent system. This situation also bears the issue of locality of changes, because the more interactions with other components (or aspects) the developer has to know in order to understand the system, the more complex the maintenance of this software results.

For future work we want to apply the approach to more case studies and real systems. Also, we intend to develop a tool to support the approach. Finally, we would like to represent aspects in other UML models, to have a more complete approach.

References

Siobhán Clarke, Elisa Baniassad. (2005). Aspect-Oriented Analysis and Design the Theme Approach Series: Addison-Wesley Object Technology Series. Crawfordsville, Indiana: R. R. Donnelley, 2005.

Johan Brichau, Ruzanna Chitchyan, Awais Rashid and Theo D'Hondt. (2008). "Aspect-Oriented Software Development: An Introduction", Wiley Encyclopedia of Computer Science and Engineering, 2008.

Shaker P., Peters D. (2005). "An introduction to aspect oriented software development", In The 6th AOSD Modeling With UML Workshop, October 2005.

Aspect-Oriented Software Development, (2009). [Online] Available: http://www.aosd.net/wiki/index.php?title=Glossary (November 11, 2009).

Structured Systems Analysis and Design Method. (2009), Wikipedia, the free encyclopedia. [Online] Available: http://en.wikipedia.org/wiki/Structured_Systems_Analysis_and_Design_Method (2009).

A. Rashid, A. Moreira, and J. Araujo. (2003). "Modularisation and Composition of Aspectual Requirements," presented at 2nd International Conference on Aspect Oriented Software Development, Boston, USA, 2003.

A. Rashid. (2005). Aspect-Oriented Requirements Engineering and Architecture Design. [Online] Available: Early Aspects: http://www.early-aspects.net/ (2005).

Nam Pyo Suh. (2007). Axiomatic Design - Advances and Applications. New York : Oxford University Press Chapter 5, pp. 239-298.

Georg, G. and Ray, I. and Anastasakis, B. and Toahchoodee, M. and Houmb, S.H. (2009). "*An aspect-oriented methodology for designing secure applications.* Information and Software Technology", 2009.

"A Retroactive Study of Aspect Evolution in Operating System Code." Yvonne Coady and Gregor Kiczales. In Proceedings of the International Conference on Aspect-Oriented Software Development 2003.

Object oriented analysis and design. (2009). Wikipedia, the free encyclopedia. [Online] Available: http://en.wikipedia.org/wiki/Object-oriented_analysis_and_design (December 20, 2009).

Object Oriented Modeling. (2009). Wikipedia, the free encyclopedia. [Online] Available: http://en.wikipedia.org/wiki/Object-Oriented_Modeling (December 21, 2009).

Jon Holt Institution of Electrical Engineers. (2004). UML for Systems Engineering: Watching the Wheels IET, 2004 ISBN 0863413544.

UML Superstructure Specification Version 2.2. OMG, February 2009.

John Hunt. (2000). The Unified Process for Practitioners: Object-oriented Design, UML and Java. Springer, 2000. ISBN 1852332751.

Bieberstein et al. (2009). Executing SOA: A Practical Guide for the Service-Oriented Architect (Paperback), IBM Press books, 978-0132353748.

Marc lives. (2010). Aspect Oriented Programming / Aspect Oriented Software Design. [Online] Available: www.codeproject.com/kb/architecture/aopByMarcClifton.aspx (February 12, 2010).

Parnas, D.L. (1972). On the Criteria To Be Used in Decomposing Systems into Modules, in: Communications of the ACM, December 1972, Vol. 15, No. 12, 1053-1058.

A b c Filman, R. and D. Friedman. (2000). "Aspect-oriented programming is quantification and Obliviousness." Proceedings of the Workshop on Advanced Separation of Concerns, in conjunction with OOPSLA'00 (2000).

Levitt, D. (2000). Introduction to Structured Analysis and Design Retrieved.

Mohammad.R. (2004). Issues of Structure VS Object–Oriented Methodology of Systems and Design (Volume V, No 1) Retrieved from University Houstan-Clear Lake.

Mylopoulos J. (2000). Structured Analysis and Design Technique (SADT).

Object-oriented Analysis and Design. (2009). Wikipedia, the free encyclopedia. [Online] Available: http://en.wikipedia.org/wiki/Object-oriented_analysis_and_ design (June 7, 2009).

Top Dow Design. (2009). Wikipedia, the free encyclopedia. [Online] Available: http://en.wikipedia.org/wiki/Top-down_design (Sept 11, 2009).

W. John, Jackson. R, D. Bur. Stephen. (2005). Systems Analysis and Design in a Changing World Retrieved.

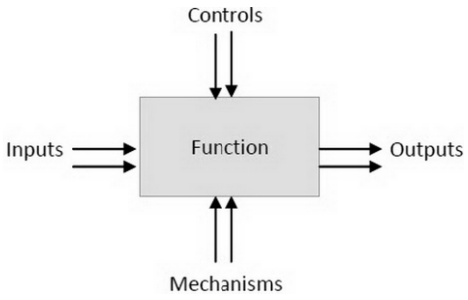

Figure 1. SADT basis elements.

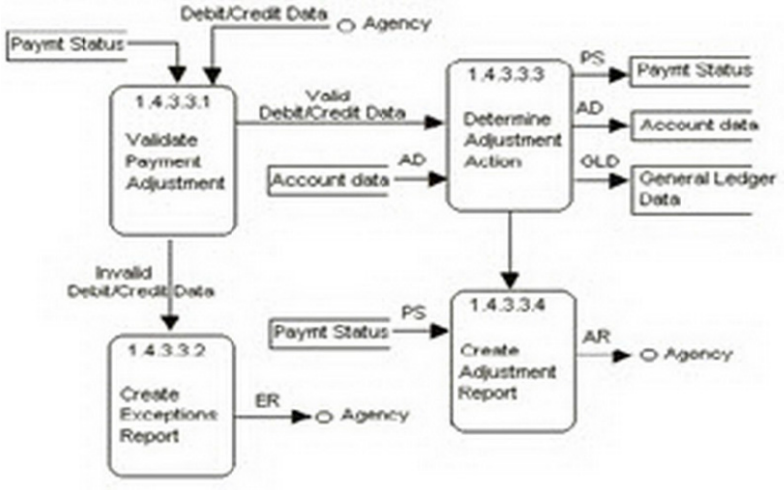

Figure 2. DFD sample.

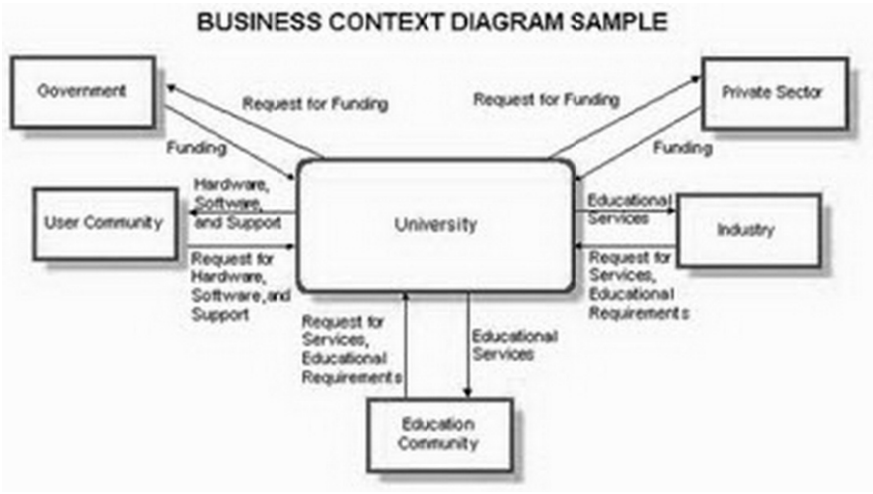

Figure 3. Business Context Diagram Sample.

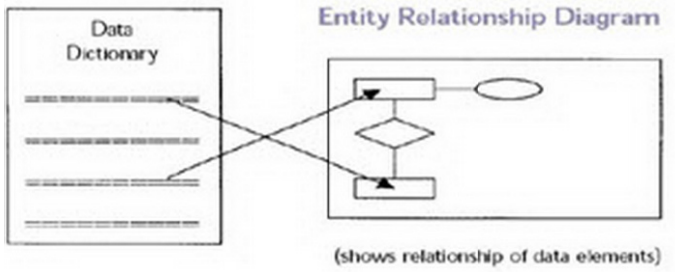

Figure 4. Entity Relationship Diagram.

ISSN 1913-8989 E-ISSN 1913-8997

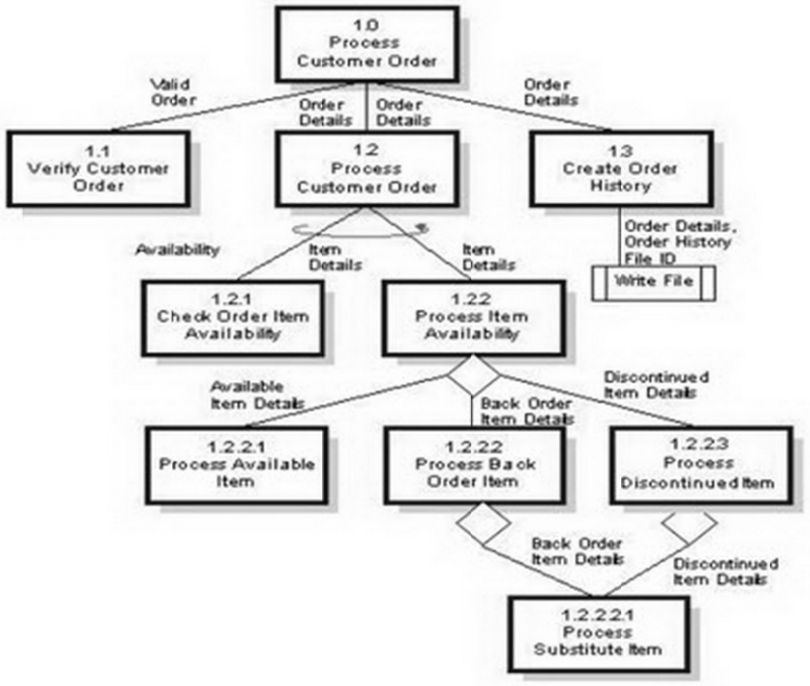

Figure 5. Structure Chart Sample.

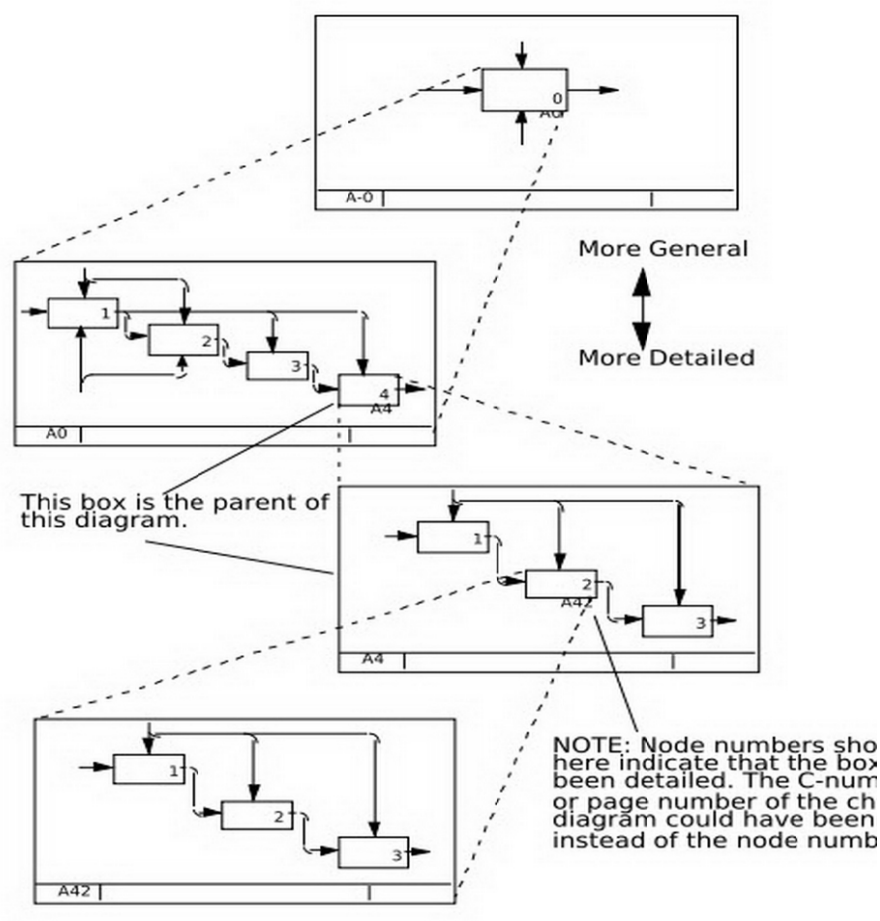

Figure 6. Decomposition Sample.

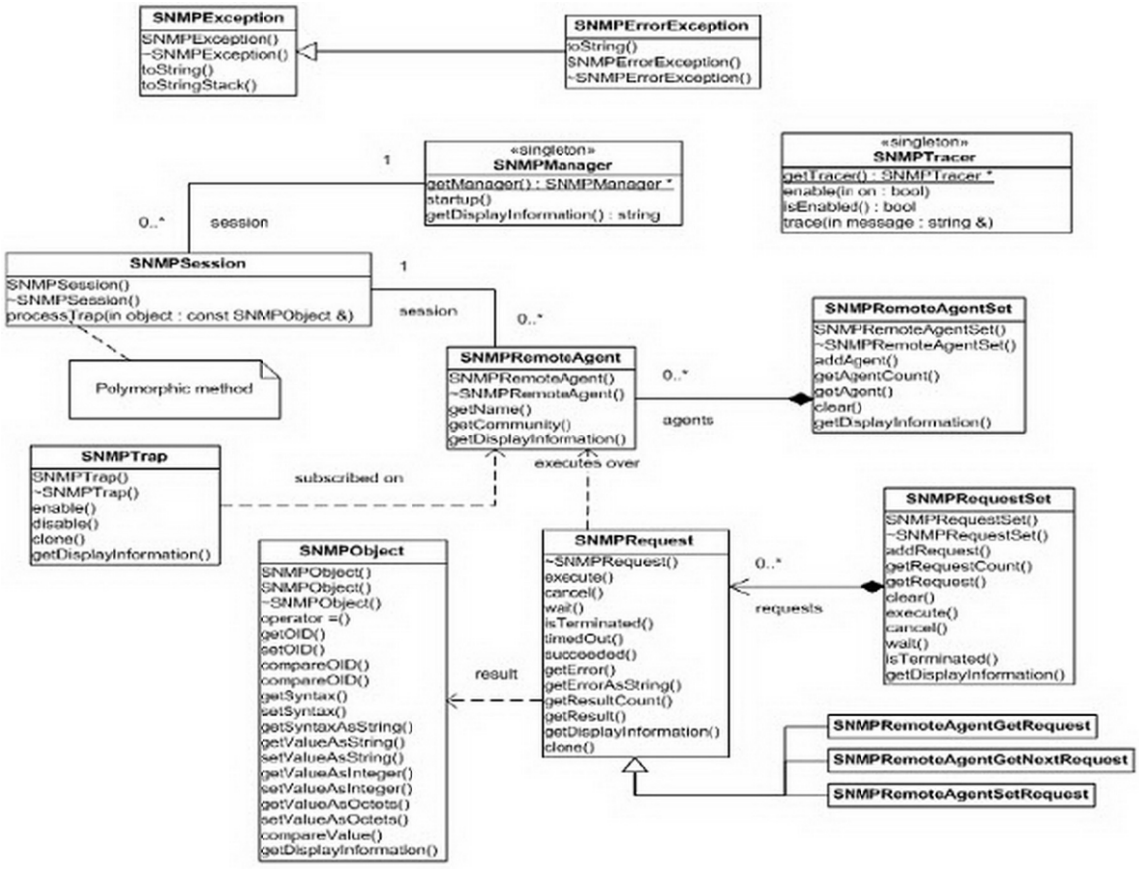

Figure 7. Class diagrams sample.

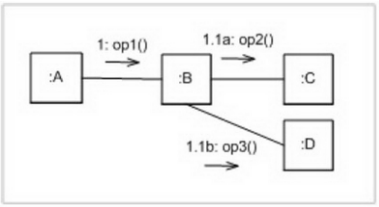

Figure 8. Communication diagram sample.

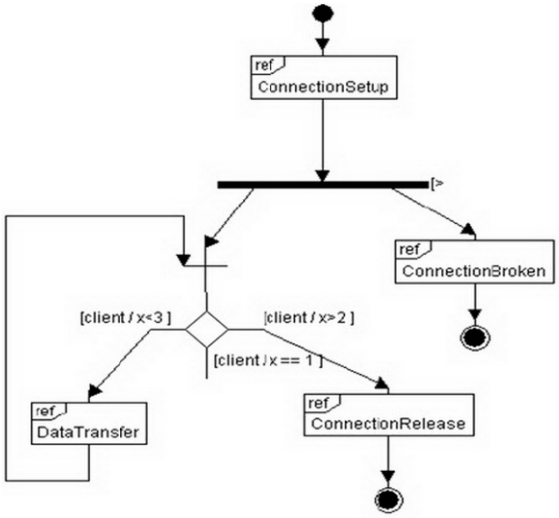

Figure 9. Interaction overview diagram sample.

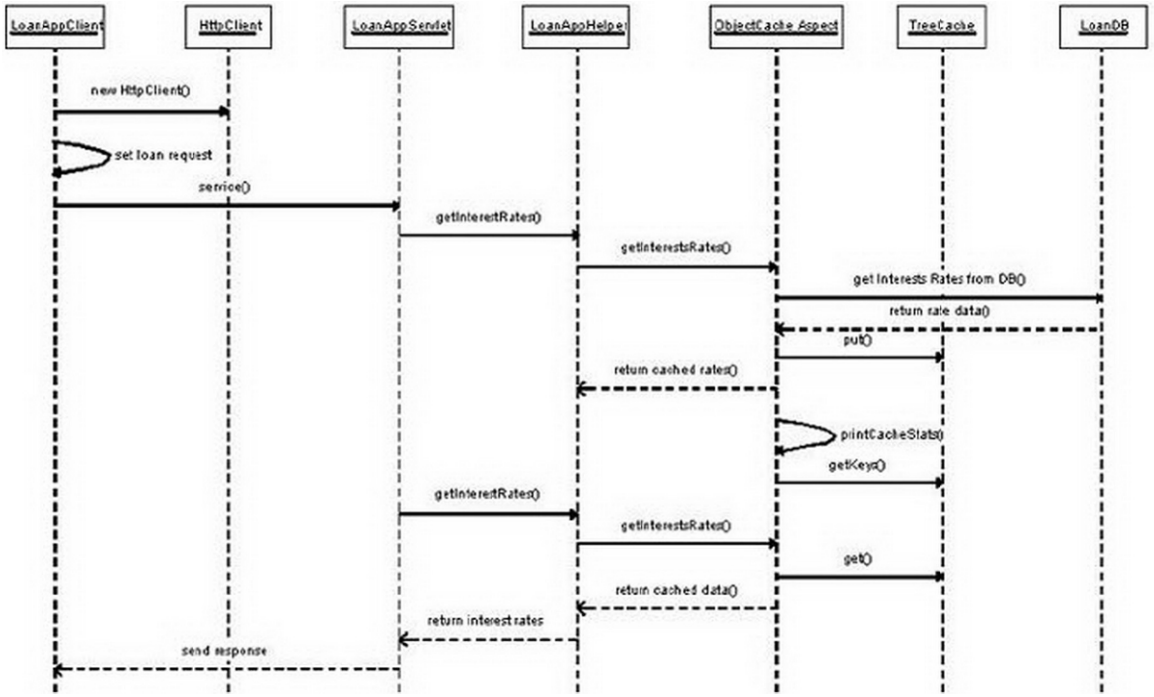

Figure 10. Sequence diagram sample.

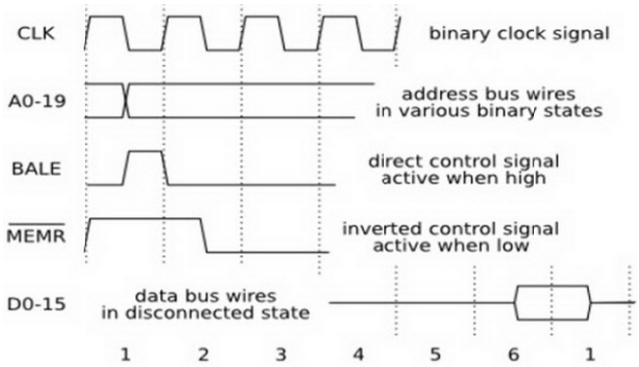

Figure 11. Timing diagram Sample.

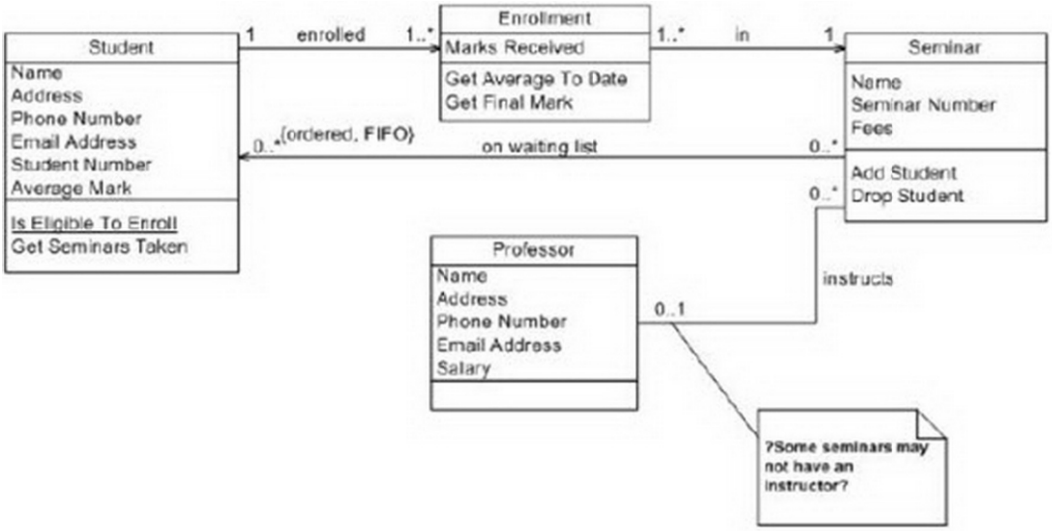

Figure 12. Class diagram derived from conceptual diagram

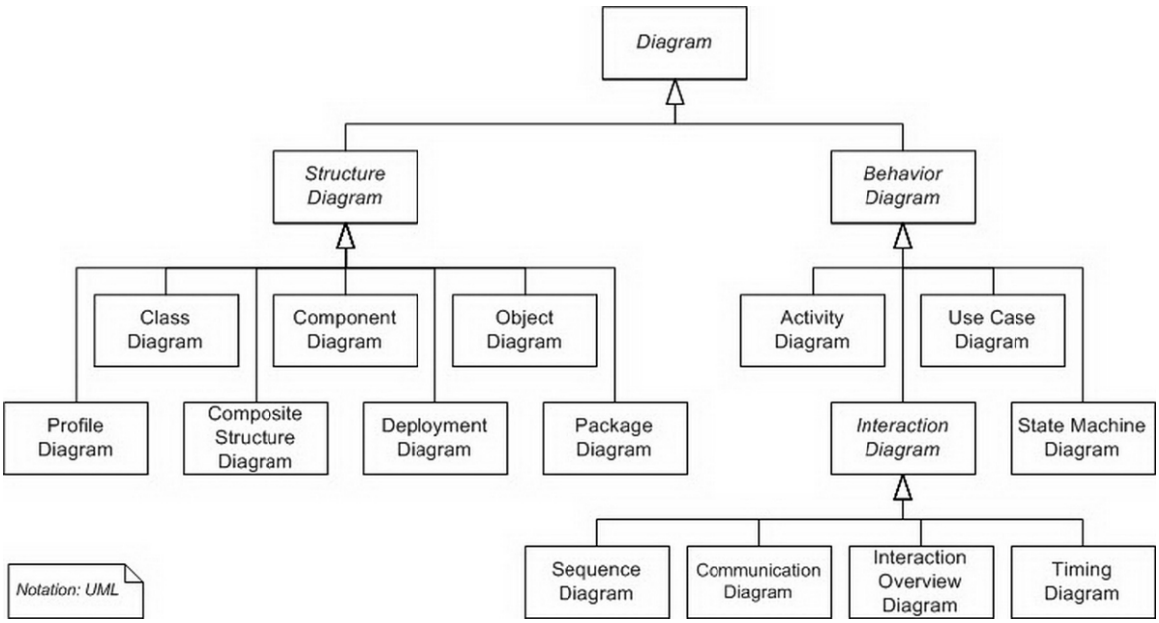

Figure 13. UML diagrams.

Figure 14. Class diagram.

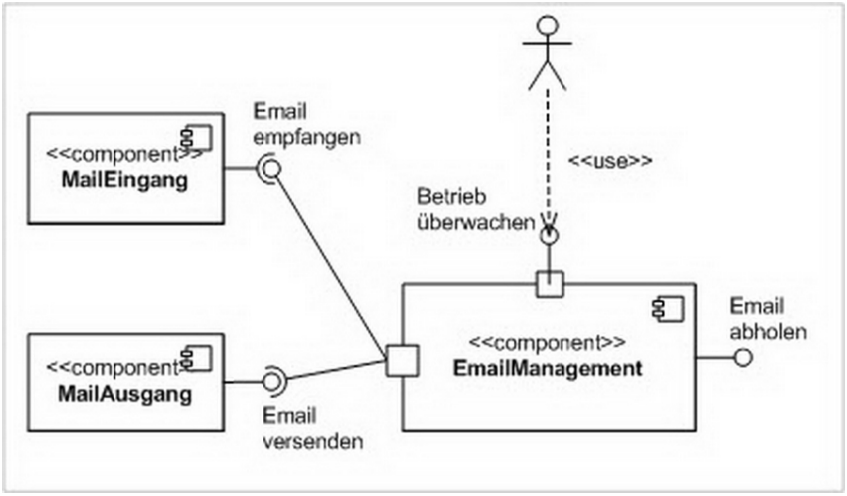

Figure 15. Component diagram.

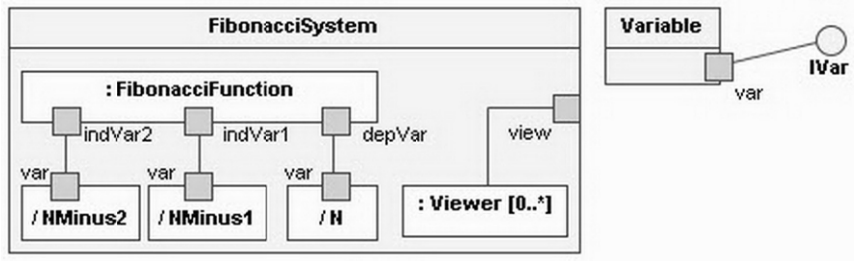

Figure 16. Composite diagram.

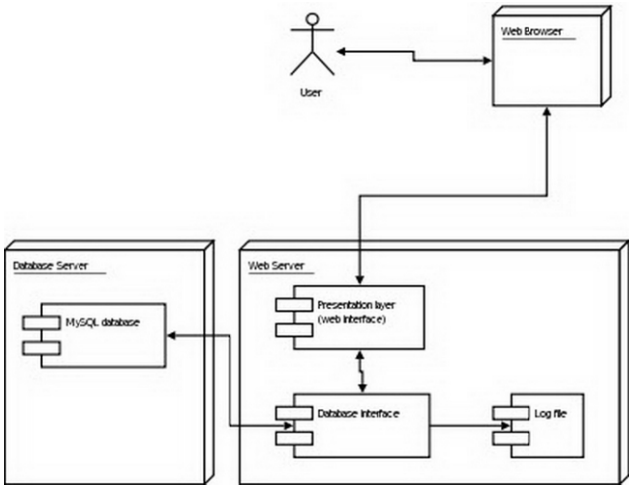

Figure 17. Deployment diagram.

Figure 18. Object diagram.

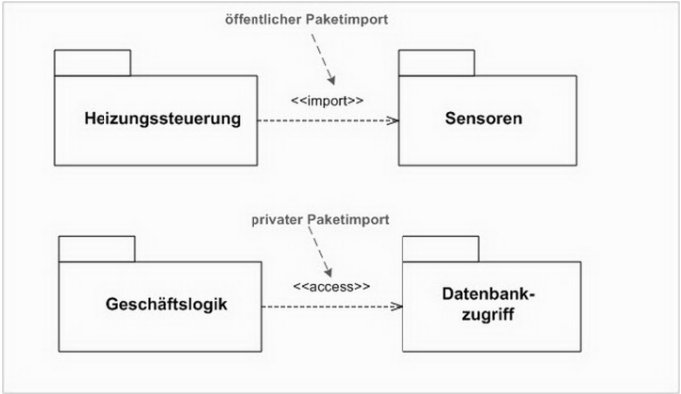

Figure 19. Package diagram.

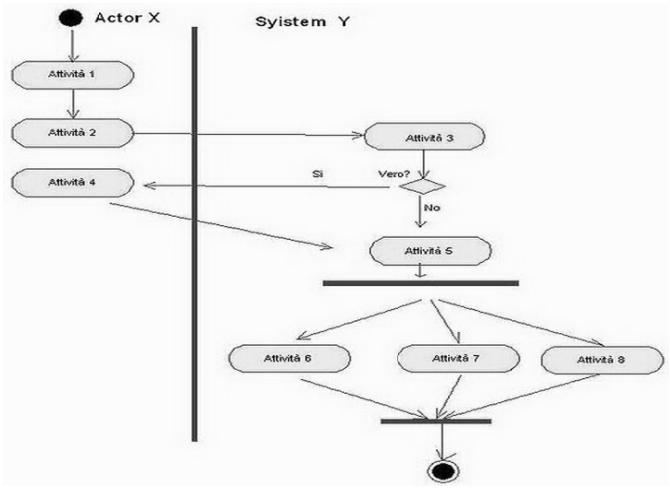

Figure 20. Activity diagram.

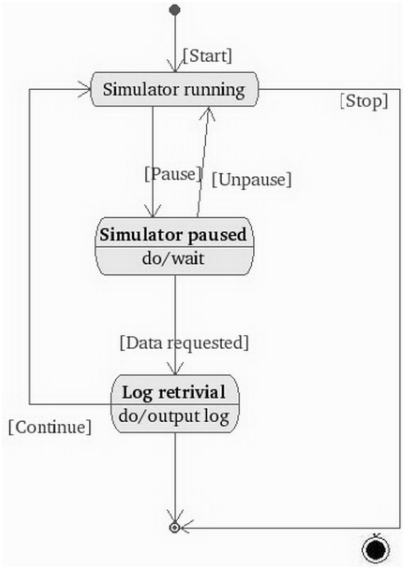

Figure 21. State machine diagram.

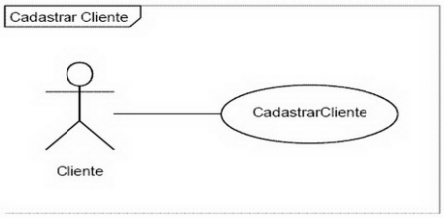

Figure 22. Use case diagram.

Figure 23. Communication diagram.

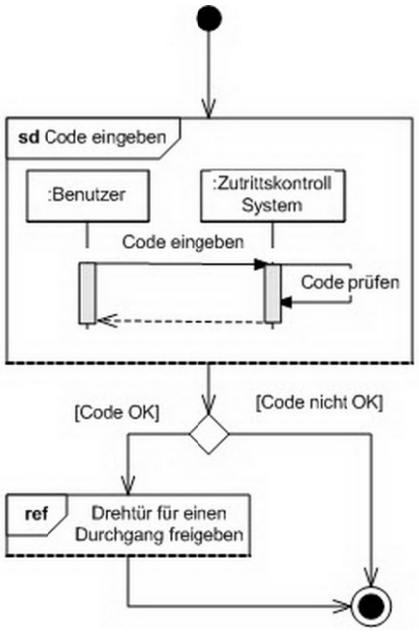

Figure 24. Interaction overview diagram.

 ISSN 1913-8989 E-ISSN 1913-8997

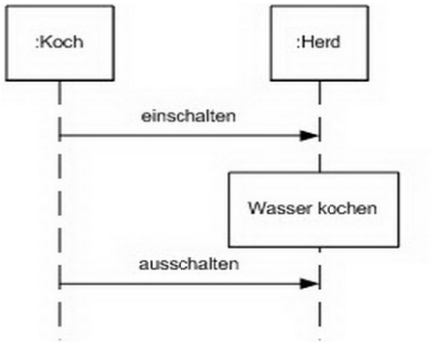

Figure 25. Sequence diagram.

```
void transfer(Account fromAccount, Account toAccount, int amount)
 throws Exception
{
Authentication Concern   if (!getCurrentUser().canPerform(OP_TRANSFER)) { throw new SecurityException(); }
Transaction Concern      Transaction tx = database.newTransaction();
                         try {   if (fromAccount.getBalance() < amount)
                                 {   throw new InsufficientFundsException();   }
                                 fromAccount.withdraw(amount);
                                 toAccount.deposit(amount);
Transaction Concern              tx.commit();
Logging Concern                  systemLog.logOperation(OP_TRANSFER, fromAccount, toAccount, amount);
                             }
                         catch(Exception e)
Transaction Concern      {   tx.rollback();
                             throw e;   }
}
```

Figure 26. Problem before SOC.

```
void transfer(Account fromAccount, Account toAccount, int amount)
    {
     if (fromAccount.getBalance() < amount)
        {
            throw new InsufficientFundsException();
        }
     fromAccount.withdraw(amount);
     toAccount.deposit(amount);
    }
```

Figure 27. Problem after SOC.

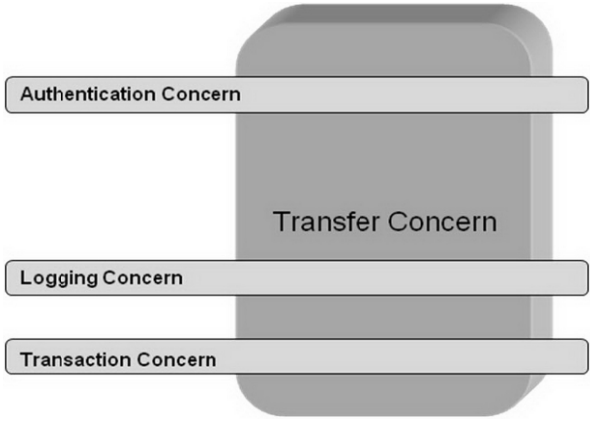

Figure 28. Transfer Concerns.

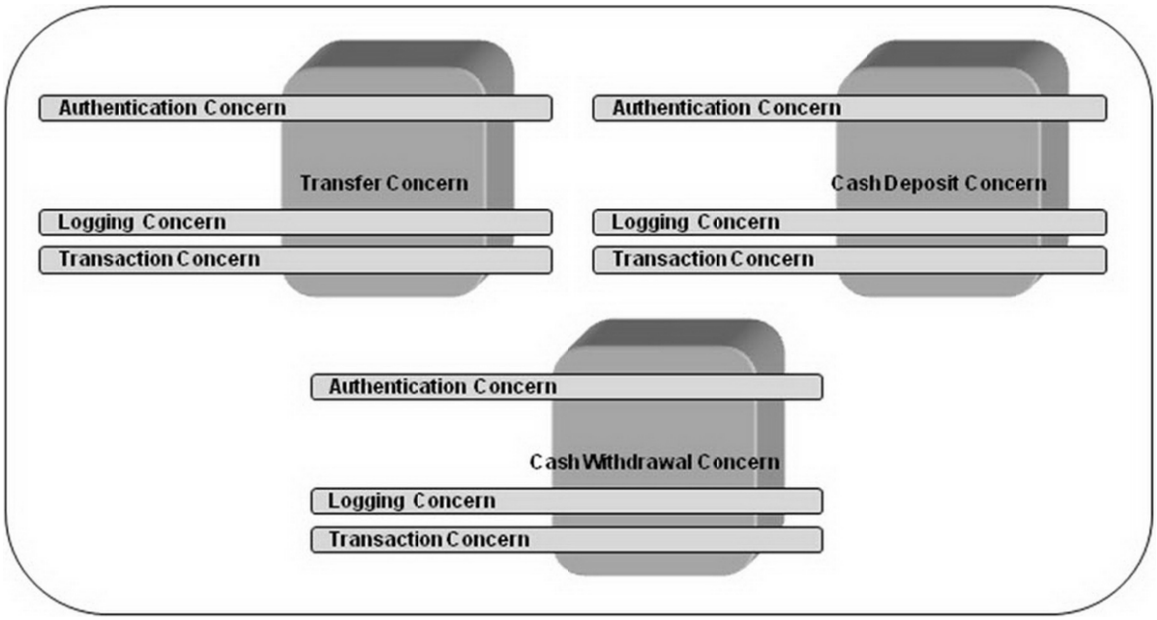

Figure 29. Banking System with SOC.

```
aspect authentication    //this is the aspect
{   before: call (public void Transfer (..))   // this is a pointcut
    {     // this is the advice
          int tries = 0 ;
          string userPassword = Password.Get ( tries ) ;
          while (tries < 3 && userPassword != thisUser.password ( ) )
          {     // allow 3 tries to get the password right
                tries = tries + 1 ;
                userPassword = Password.Get ( tries ) ;
          }
          if (userPassword != thisUser.password ( )) then
                    //if password wrong, assume user has forgotten to logout
                    System.Logout (thisUser.uid) ;
    }
} // authentication
```

Figure 30. Aspect Structure (authentication code sample).

 ISSN 1913-8989 E-ISSN 1913-8997