Facilitating software development using UML models

Rodney Gunik

Abstract

The Unified Modeling Language (UML) is a visual language which aids in analysis and design of software systems using the object-oriented approach. The interesting connections between the UML models and the framework the models provide are discussed to aid interpretation of problem domains by computing science students thereby enabling them to develop conceptual models visually to facilitate software development. A theoretical student management system is used to demonstrated the connections between the models and how they could be used for software designs.

Keywords: software development, systems analysis and design, Unified Modelling Language, use case, domain class, activity diagram, system sequence diagram, sequence diagram, design class diagram, object-oriented approach.

Introduction

Most computing science units in the Mathematics and Computing Science program at Divine Word University are based on software development. Students studying these units begin coding by learning how to use proper syntax and semantics for a popular programming language. However, understanding the problems of software development is a skill required in the development of software systems. While this is true for software development, Unified Modeling Language (UML) models could be used to analyze and design a software before the actual coding.

Students require a well-presented knowledge of how to convert their software ideas into coding. As a way of introducing software analysis and design, this paper aims to demonstrate the applications of UML models and how they relate to the analysis and design of software with the aid of a theoretical example for a student management system (SMS). The paper begins with the conceptual design by identifying the functional requirements using the use case diagram (UCD). Further, domain classes are developed, which define the structure of system objects. Activity diagrams (AD), system sequence diagrams (SSD) and sequence diagrams (SD) are developed to model the internal behavior of the system which serves as a portal for identifying method signatures. Finally, design class diagrams (DCD) are developed using the domain models and method signatures.

Use case

"A use case is an activity that a system performs in response to a request conveyed by a user" (Satzinger et al. 2012, p. 69) who is normally a person using the system. That user is called an actor because the system could also receive request from other systems. The actor is always situated outside the automation boundary of the system which is part of the system's manual region (Figure 1). There are two techniques for identifying use cases (user goal and event decomposition).

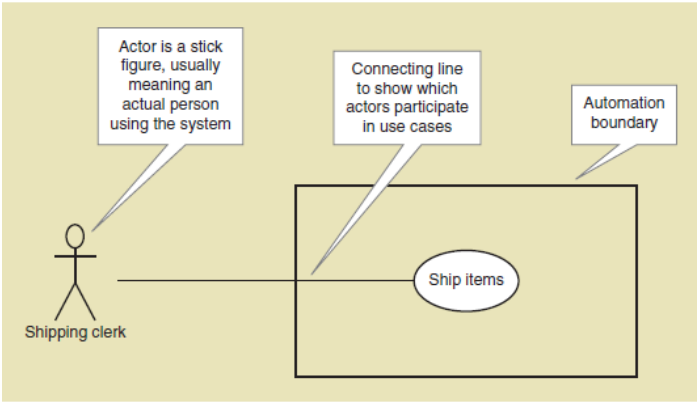

Figure 1 A use case diagram where a user is accessing the *ship item* use case from outside the automation boundary (Source: Satzinger et al. 2012, p. 81).

With the user goal technique, "users are guided through a series of specific questions that identify the tasks users would like to accomplish with the help of a system" (Satzinger, et al. 2012, p. 69). This technique is usually applied during interviews when gathering requirements from users.

"On the other hand, the event decomposition technique is more comprehensive because the process begins by identifying all the events that surrounds the system and triggers the system to respond" (Satzinger, et al. 2012, p. 70). The response executed by the system as a result of an event leads to a use case.

System requirements (functional and non-functional) (Table 1) are usually gathered using the information gathering techniques (e.g. interviews) but USD can be used for the requirement modelling process. The requirements could be used to define the activities supported by the system.

Table 1 Functional and non-functional requirement where the functional requirement has one category compared to the non-functional requirement (Satzinger, et al. 2012, p. 43).

Requirement categories	FURPS + categories	Example requirements
Functional	Functions	Business rules and processes
Nonfunctional	Usability Reliability Performance Security + Design constraints Implementation Interface Physical Support	User interface, ease of use Failure rate, recovery methods Response time, throughput Access controls, encryption Hardware and support software Development tools, protocols Data interchange formats Size, weight, power consumption Installation and updates

"The functional requirements are the activities specified by an organization's business rules and other functionalities that can be accomplished by the system" (Satzinger, et al. 2012, p. 42). The non-functional requirements refer to the attributes that are used to measure the quality of a system such as performance, security and infrastructure. Although its discussion is limited in this article, the non-functional requirement is as critical as the functional requirement.

"Since a use case defines functional requirements by identifying the actions performed by the users" (Satzinger, et al. 2012, p. 69), these actions become obvious in a use case description (Table 2). The verb portion of a use case identifies the actions taken by the actor to achieve some specific goal. The noun portion identifies the system's object in the problem domain, which facilitates the interpretation of real-world objects in the system.

Table 2 The use case description for student management system

Student Management Subsystem	
Use case	**User/Actor**
Enroll Student	Registrar, School Secretary, Student (Self-enrolment)
Search Student	Registrar, School Secretary
View Classlist	Registrar, School Secretary, Teachers,
Create gradebook	Teachers
Enter grade	Teachers
Print Classlist	Teachers, Registrar, School Secretary
Update Student	Registrar, School Secretary

The process of applying use cases to system analysis and design can be demonstrated using a SMS (Table 2) and the event decomposition technique. This technique can be used to identify all external events that occur during the process from adding a student to the system to managing student data. These

events require an action to be taken by an actor from outside the automation boundary such as enrolling students, lookup student, view the student name list, create grade book, manage student grades and update student information. The actors who perform the use cases are listed under the right column and use cases under the left column (Table 2). The use cases are also written in the verb-noun form in order to facilitate the development of domain classes compared to a UCD which are used to develop ADs (Figure 4). The UCD is a visual representation of the use cases denoted by ovals and lines indicating the association of actors to each use case. This leads us to the domain class model which is designed using the nouns from the use case description.

Domain class

"Domain classes are also known as data entities or models that represent objects of a system" (Satzinger, et al. 2012, p. 92). They are designed to capture and represent data for the objects during a system's operation. The problem domain mentioned in the previous section is a specific area in which the scope of a system defines, therefore, real-world objects are captured and modeled for that area (Figure 2).

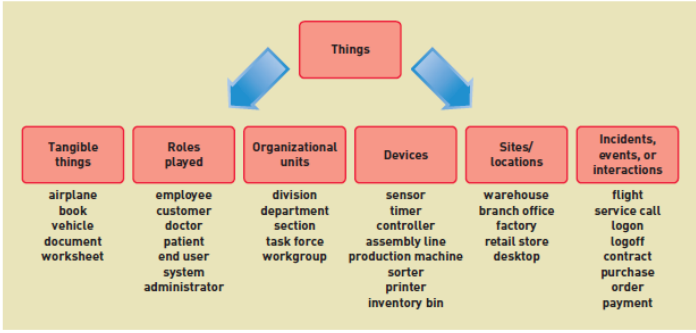

Figure 2 The category of things or objects in the problem domain is a concept used to identify problem domain objects (Satzinger, et al. 2012, p. 93).

There are two techniques used for identifying objects in the problem domain (brainstorm and noun technique) (Satzinger, et al. 2012).

The brainstorming technique is used to identify objects through interviews designed to discuss the types of objects and processes the users may encounter during business operating hours. The noun technique is used to list all the nouns a user mentions unintentionally while discussing the process and goals of the system. Domain classes can also be identified from the nouns that compose a use case description. A list of nouns can be compiled from the descriptions during the investigation of use cases. Seizing this approach would save time and resources from conducting multiple interviews (Satzinger, et al. 2012).

As mentioned previously, domain class models represent objects of the real world, therefore, every real-world object has properties and behaviors. An object's property is also known as attribute which describes the individual characteristic of objects that differentiate one from another by determining state, appearance and other qualities of the object (Lemay et al, 1996). An object's behavior is anything the object does, which involves actions that are stimulated by external factors (Marian Webster, 2017). The only way an object could behave is to function for itself or other objects. However, during the development of domain class models, emphasis is placed on attribute rather than behavior because behavior of objects is made available when interaction models are developed.

Classes provide a template for an object and can generate multiple instances during runtime. Therefore, during the process of developing domain classes, attributes of each of the objects in the problem domain are identified and applied to the class model to develop domain class diagrams. Relationships between objects of the problem domain are also indicated in a domain class model (Figure 3).

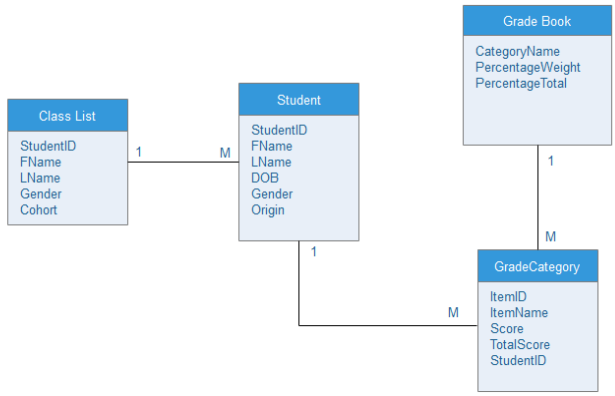

Figure 3 The domain classes for the student management system which are developed from the use case description.

Using the nouns in a use case description to build domain class models can be applied to our theoretical example of the SMS. Every noun in the use case description (Table 2) is listed. Careful consideration should be given if the list of use case descriptions is lengthy because nouns can have duplicates which might duplicate the domain classes such as Student and Classlist in the table. Attributes that match the noun title are listed accordingly (Figure 3) with the unique attribute such as Student ID and Item ID to be identified for each object. Then other attributes such as name, gender, score and DOB are added.

Relationships can be identified using logic and natural language. There can be many students in one class list, denoted by (1 and M), one student can have

many grade items (1 and M) and many grade items can be found in one grade book (M and 1). Domain class models provide an overview of the various objects in the problem domain and how the objects are related to each other. This leads to the discussion of the activity diagram.

Activity diagram

"An activity diagram provides investigation into the use case diagram by examining the sequence of activity between an actor and use case" (Satzinger et al. 2012 p. 57). It is used to identify the sequence of steps that are performed to complete a use case. In a UCD, lines indicate actor's collaboration with a use case (Figure 4). However, more information is required to provide details about the interaction between the actor and use case as well as how the action is conducted in sequences. These developments are initiated so that the use case show the interactions and flow of the action taken by the actor to invoke the system to respond with some result (Figure 5).

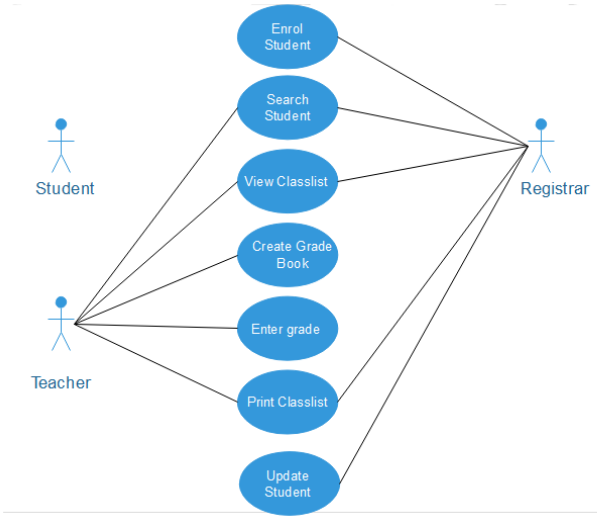

Figure 4 A use case diagram for student management system with two actors in the role of a teacher and registrar. The student role does not associate to any use case.

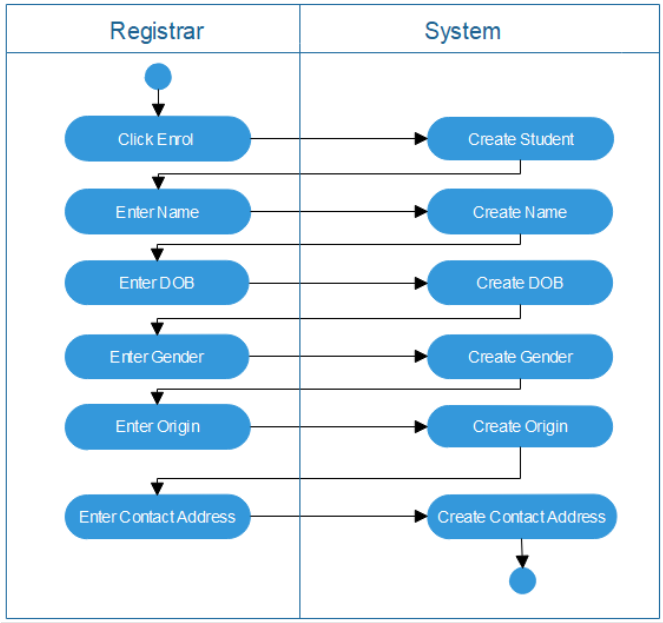

Figure 5 A simple activity diagram for *enrol student* use case which is developed from the use case diagram.

The oval in an AD represents an activity in a workflow (Satzinger, et al. 2012). The diamond represents a decision point at which the process can take either paths available. The sequences of activities are represented by the arrows and the black circles denote the beginning and ending of the workflow. “The heavy solid line is called a synchronization bar, which split the flow of activities into multiple concurrent path or recombines them at the end of the flow” (Figure 6) (p. 57). A swimlane is a column in which each entity that participates in the flow of activities are located.

The ADs for each use case in a UCD are developed so that every activity flow is modelled. In order to develop activity diagram for the SMS, the UCD in Figure 4 is used to select the use cases. Each use case is developed into an activity diagram. Figure 5 shows the activity diagram for the *enroll student* use case. Consider the process when a registrar is enrolling a student either automatic or manual. The process will be executed in sequence by first entering the students name, date of birth, gender, origin and more information about the student. The AD indicates sequence of multiple actions between the actor and the system, however, in a real system the sequence from *Enter Name* to *Enter Origin* in Figure 5 could be executed once.

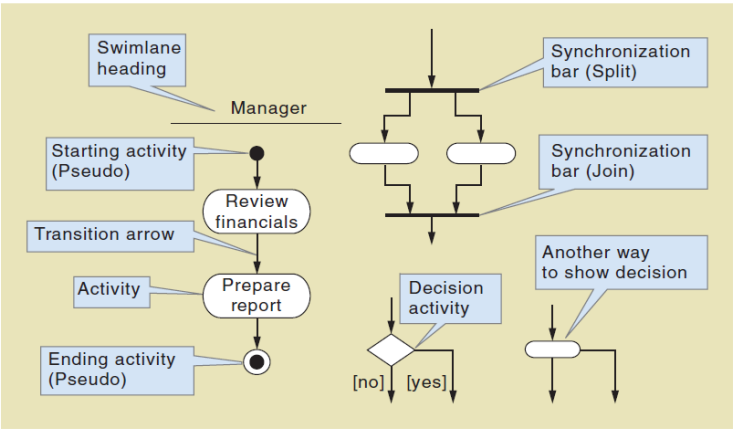

Figure 6 The basic symbols for an activity diagram where each of the symbols are labelled with quotes for clarification (Satzinger et al. 2012, p. 58)

The arrows indicating the sequence of actions that goes back and forth the swimlane in the AD require further details. It is now essential to discuss the SSD to provide further details about the sequence.

System sequence diagram

"The flow of information between the objects and actor is achieved through messages sent from actor to the system or between internal objects" (Satzinger, et al. 2012, p. 126). Further developments are done to the AD, which include the visibility of information that passes between the user and system. A SSD as mentioned above, shows the flow of information such as inputs, outputs and interaction between the actor and system.

A SSD employs similar symbols as used in the UCD. A stick figure is used to represent an actor which interact with the system, which is viewed as a black box bearing the label System (Satzinger, et al. 2012).

A SSD contains two lifelines representing the actor and system. Lifelines are indicated by dashed lines extending vertically from the communicating entities to indicate the activation time for each entity. Arrows are used to indicate the direction of the messages communicated between the actor and object of the system (Figure 7). The sequence of messages defined on the SSD is read from the top to bottom.

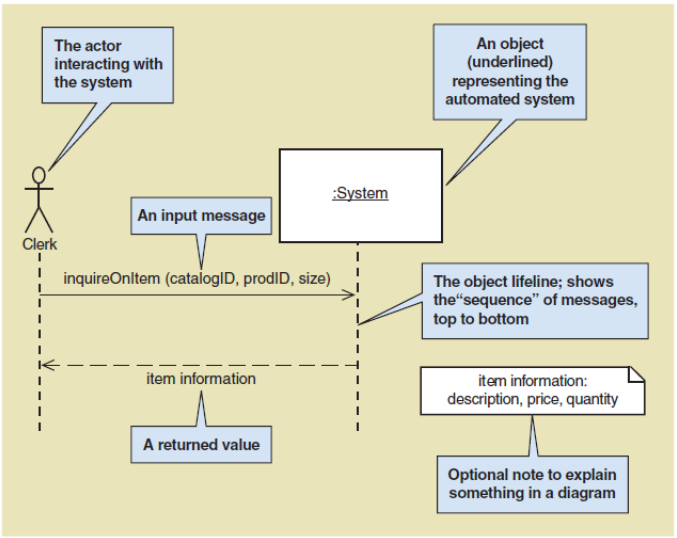

Figure 7 A diagram showing basic symbols for system sequence diagram where the actor is a clerk and system is seen as a block box. Each symbols are labelled with quotes for clarification (Satzinger et al. 2012, p. 127).

"The information that passes between the system and actor for execution of a task was discovered by the use cases discussed earlier. Labels are placed before the input messages to describe the type of input and enclosed with parenthesis" (Satzinger, et al. 2012, p. 127). The input messages are composed of attributes of the noun found on the message name. The message name uses the verb-noun form to project the purpose of the message. The returned messages are displayed using a different syntax because they are only for outputs which do not require a service to be done by the user.

To further develop the models for the SMS, the activity diagram is used to develop a SSD (Figure 8). The input messages are the attributes of the students identified in the domain class model because every data entry is specific for each attribute. The actor is situated on left lifeline and :System on the right. The return message could be displayed as the output on the screen indicating the response from the system. Details excluded in the earlier models can be added and updates can be done to the previous models.

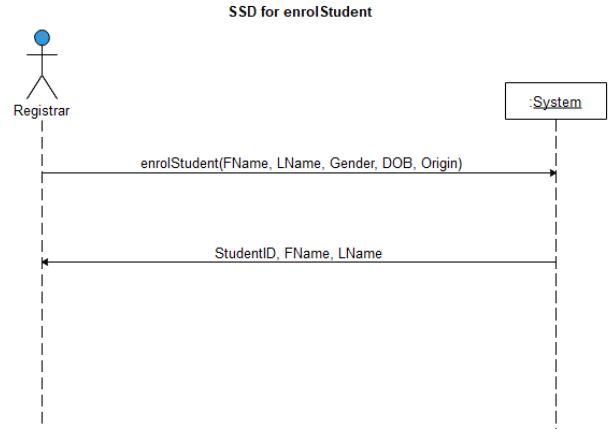

Figure 8 A system sequence diagram for enroll student use case where the actor is the Registrar and the system is view as a black box.

A SSD is further developed to show that all objects that function together to perform a use case are specified. The :System object, which is seen as a black box, is usually expanded using the multilayer system design. This is leads to the discussion of sequence diagrams.

Sequence diagram

Using multilayer design, a SD is developed by expanding the :System in the SSD into system of objects into three layers (view layer (VL), domain layer (DL) and data access layer (DAL)) (Satzinger, et al. 2012).

The VL contains objects from the user interface design, which is primarily responsible for formatting and presentation data to the users. Inputs are also entered, edited and forwarded from the VL of the system. The DL contains the objects in the problem domain, which is also referred to as the business logic layer because it contains core objects of the use case. The DAL is responsible for connection to the database, read and send the data back to the domain objects. Whenever data require saving, the DL pass these data to the DAL for writing it to the database.

The SD has symbols for objects, actor, lifeline and activation, messages and arrows (Figure 9). The objects on a SD is a rectangle with the object's name and arranged in a timely sequence with reference to the lifeline and activation time. The actor is a stick figure, which represents the role of the entities that affects the system. The messages maintain the same syntax from SSD and can be communicated between the actor and objects or between objects within the system.

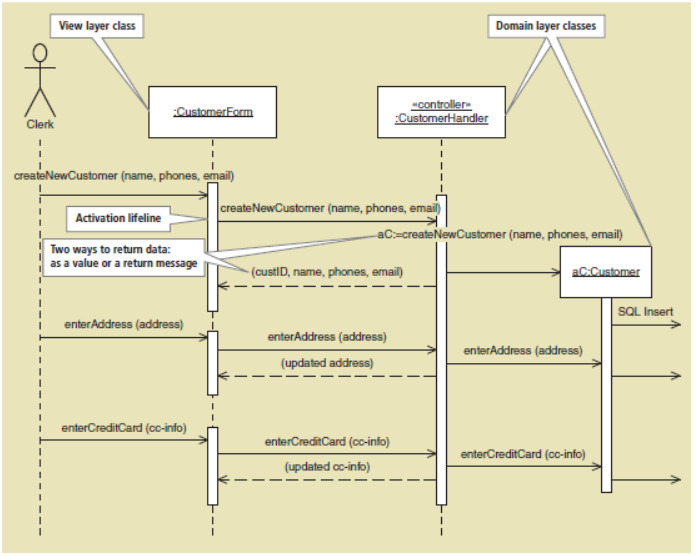

Figure 9 A sequence diagram for a create customer use case that demonstrate basic symbols for a sequence diagram. Again the Clerk is the actor in this sequence diagram (Satzinger, et al. 2012, p. 334)

"Lifelines are dashed lines extending from the lower end of objects and actor which represents the existence of the entities at a particular time" (Satzinger, et al. 2012, p. 334). They indicate the existence of objects by terminating the dashed line if destroyed, starting a new dashed line if created or continuing the line if the object continues to exist. Activation is shown by a tall thin rectangle which indicates that the object is actively existing and duration of the activation is indicated by how tall the rectangle extends along the lifeline. The incoming message indicates the object is performing some function, therefore, activation is indicated by extending right after the incoming message.

In a SD each message has a source and destination object (Satzinger et al. 2012). When the source sends a message, the destination should be prepared to accept the message, which will invoke the destination to initiate some activities. The process that initiates an activity on the object is through calling a method on that object. Methods specify how the objects behave during their lifetime. The activities that are called on the objects are known as *method signature* and are useful in the development of the DCD.

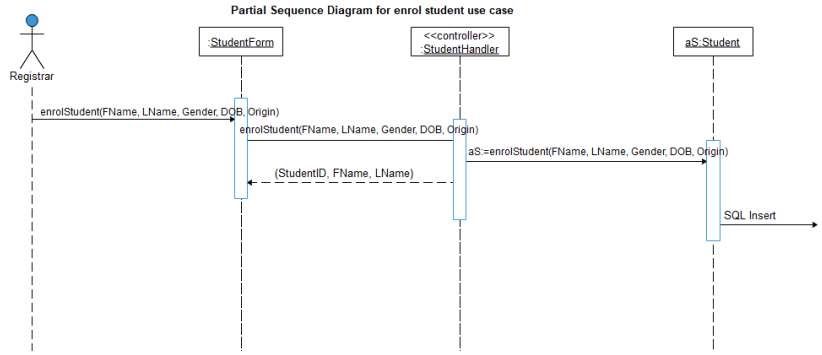

Figure 10 A partial sequence diagram for the *enroll student* use case where only the objects from the view layer and the domain layer are shown. Messages are received at the StudentForm and passed across the internal objects.

The System object from the SMS is expanded by adding three additional lifelines and labels (rectangles) representing each layer of objects. More objects can be added when a use case indicates interaction between many objects. The message syntax remains the same except the introduction of reference variables. Figure 10 is called a partial SD because the objects from the DAL is not indicated on the diagram. The variable *aS* is a reference variable for the student object while the activation for each object in the system is indicated using the tall thin rectangles. The controller object is introduced to operate has a switch by accepting and directing incoming messages to the appropriate DL object. The DCD will now be discussed to show how behaviors of objects are realized from the SD.

Design class diagram

Design class diagrams (DCD) are usually developed parallel to the interaction diagrams, which are accomplished by updating the domain classes with a *method signature* to the lower compartment of the domain class models. Method signatures are devised based on the details realized from the sequence diagram. Most classes contain only three types of methods (constructor, data-get, data-set, and use case-specific).

The constructor method is a special type of method which is called to create an object (Satzinger et al. 2012). It could also accept arguments and set required member variables for the new object. Data-get and data-set methods are public methods, which allow new values to be assigned to private variables and make the values accessible. The *get* method allows access to values of variables while *set* method assigns value to the private variables. The use case-specific methods are driven by the use cases and determined by the behavior of objects, which are realized from the SD. These methods are displayed as signatures in the DCD (Figure 11).

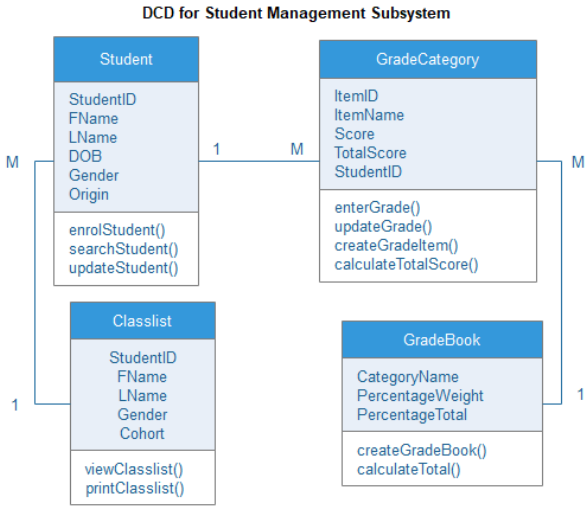

Figure 11 The design class diagram for student management subsystem where method signatures are identified from the interaction diagrams are indicated at the lower compartment.

In light of OOA for software design some of the major components of the models such as attributes, behaviors, objects and classes are discussed. This approach focuses on capturing the attributes and behavior of objects which will facilitate software development by converting ideas into coding (Dennis et al. 2015, p.19).

Since the structure of a DCD model is known we can now develop a DCD for the SMS (Figure 11). The method signatures for the class *Student* have been identified by identifying which messages will be sent to the Student object as incoming messages. In this case, *enrolStudent* is the incoming message as shown in the SD in Figure 10. In addition, method signatures for the classes including *Classlist*, *GradeCategory* and *GradeBook* were identified as possible incoming messages and indicated at the lower compartment in the DCDs.

After developing DCDs that show relationships among classes, attributes of the classes and method signatures, the model provides an excellent documentation and can serve as a blueprint for beginning the programming of the system.

Conclusion

This paper has attempted to provide computing science students with skills in the analysis and design of software systems using the UML models. It has provided information on how investigation is conducted in the problem domain, and models were developed using each of the diagrams. For each model, applications were demonstrated and related to the processes of analysis and design of the SMS. Demonstrations for use cases, domain classes, activity diagrams, system sequence diagrams, sequence diagrams and design class

diagrams were accomplished using a theoretical example of the SMS which has identified the process of converting a software idea into coding.

References

Constructor (object-oriented programming). (n.d.). In *Wikipedia*. Retrieved August 22, 2017, from https://en.wikipedia.org/wiki/Constructor_(object-oriented_programming)

Dennis, A., Wixom, B. & Tegarden, D. (2015). *Systems analysis and design: An object-oriented approach with UML* (5th ed.). USA: John Wiley & Sons.

Lemay, L., Perkins, C. & Morrison, M. (1996). *Object-Oriented Programming and Java*. Accessed from http://www.dmc.fmph.uniba.sk/public_html/doc/Java/ch2.htm#ObjectsandClasses

Satzinger, J., Jackson, R., & Burd, S. (2012). *Systems analysis and design in a changing world* (6th ed.). Boston: Cengage Learning.

Acknowledgements

I would like to acknowledge Prof. Peter K. Anderson, Head of Departments for Information Systems and Mathematics and Computing Science for his leadership, encouragement and initial editing of this paper. I also acknowledge Mr Martin Daniel, lecturer from the same departments for his ideas, editing and reviewing of the paper. However, responsibility for any errors of fact or opinion, or infelicities of expression must remain with the author.

Author

Rodney Gunik is a tutor in the Department of Mathematics and Computing Science at Divine Word University where he holds a Bachelor Degree in Mathematics and Computing Science and specializes in Applied Mathematical Analysis and CCNA. His research interests include researching in software development, mathematics & computing science.
Email rgunik@dwu.ac.pg