Business Object-Oriented Analysis and Design (BOOAD) Methodology

Zeki Bayram
Boğaziçi University
Bebek 80815 Istanbul
Turkey
e-mail: bayram@boun.edu.tr
web: http://www.cmpe.boun.edu.tr/~bayram
tel: +90 (212) 2631500 ext. 2094

Abstract

In this paper we advocate the use of *application-type and computation-paradigm* specific, as opposed to general purpose, analysis and design methodologies for developing software systems. To motivate and justify our viewpoint, we describe an object-oriented analysis and design methodology which is tailored for software systems whose functionality is to be implemented mainly through database operations. Because business information systems are typically in this category, this methodology is given the name *Business Object-Oriented Analysis and Design* methodology, or *BOOAD* for short. The notation as well as the semantics of the notation used in BOOAD is inherited from Object Modeling Technique (OMT) of Rumbaugh, but somewhat modified through the *lifting up* of various notions from *object-oriented database systems*. These modifications include, among others, the concept of considering classes as repositories of objects, having classes answer *queries* regarding the objects they contain, the concept of a *transaction* which can be rolled back if necessary, *integrity constraints* on the states of objects, and *set-valued* and *pointer-valued* attributes of objects, as well as an *exception mechanism* which provides support for error handling and error recovery. Object communication is mainly through events in the broadcast-receive mode, but point-to-point communication

among objects and classes is also supported. A sample analysis model (student registration system) is developed to demonstrate the main concepts of BOOAD.

Keywords: Business, analysis, design, event, state transition diagram, requirements specification, class, inheritance, object-oriented

1. Introduction

Specification of *what* a software system is going to do is probably the most important phase in the development of the software system. The product of this phase is known as the *requirements specification* and is a reference point used throughout the development process of the system.

The requirements specification is generated after an *analysis* of the problem at hand. This analysis results in an *understanding* of the problem, the *identification of the main functions* to be performed by the system that will be eventually produced, as well as *how the system will interact with its environment*. Although the results of the analysis phase can be described using natural language, doing so can hinder the discovery of inconsistencies or incompleteness in the specification. Due to the impreciseness of natural language, ambiguities may also be present in the specification. Some restricted form of natural language, together with graphical notations with well specified semantics is thus preferred over unrestricted textual descriptions.

Object-oriented analysis and design (OOAD) methodologies combine textual and graphical notations in a way which allows the description complex systems naturally and as closely as possible to the problem area. This is made possible by the fact that the description of the whole system is decomposed into the description how each

individual object in the system perceives and interacts with its environment and other objects.

There is a large number of OOAD methods that have been proposed, (see for example [[1],[2], [3], [4], [5]]), and some of these are in widespread use today. The problem with these OOAD methods is that they are *general purpose*, attempting to address the needs of many kinds of applications, without any regard for the actual paradigm of computation that will be used to implement the system, or the type of the application. Consequently, for a specific type of application or computation paradigm, we can expect that only part of the notation in the method will end up being used. Furthermore, it is often quite unclear how the transition from the analysis to the design model will be made, considering the fact that the *back end* could be anything from an object-oriented database system application to a computation-intensive concurrent program running on a massively parallel machine.

We advocate instead the development and use of OOAD methods that ***do*** take into account the type of application, and the paradigm of computation that will be used to implement the application. The advantages of this approach include eliminating notation at the analysis phase that does not readily map into design notions, the enrichment of the "tools" available at the analysis level through the *lifting up* of design concepts into analysis, and diminishing the semantic gap between analysis, high-level design and detailed design.

The remainder of this paper is organized as follows. In the next section, a brief overview and main characteristics of BOOAD is given. Section 3 gives a high level description of the process of object-oriented analysis and design using BOOAD.

Section 4 describes each component of BOOAD in detail. Point-to-point communication is described in section 5. The mapping from a BOOAD model onto a relational database design is given in section 6. Section 7 gives a sample analysis model generated using BOOAD and finally section 8 is the conclusion and future research directions.

2. Main Highlights of BOOAD

BOOAD contains notions *lifted up* from the world of databases as well as some that are adapted from programming languages. Briefly, these are

- *Queries* which are used to manipulate data, a notion which is present in BOOAD in the form of *class methods*. Classes are then seen as depositories of objects (i.e. sets of instances belonging to the class), and queries on classes correspond to activations of class methods. Once queries are made available in the analysis model, they can be used inside the guards or action parts of transitions in a state transition diagram (to be explained in due course), increasing the expressivity of these diagrams.

- The notion of a *transaction* which can be rolled back if necessary.

- *Integrity constraints* on states of objects. Violation of an integrity constraint causes specified actions to be taken through the raising of exceptions (see below).

- An *exception mechanism* to handle unexpected conditions. Exceptions, attached to integrity constraints or conditions in a transition in a state transition diagram, help specify the constraints on the system as a whole in a very elegant way.

This specific set of features in BOOAD makes possible both *point-to-point* and *broadcast-receive* kind of communication among objects. These will be explained shortly in the ensuing sections.

3. The Process of Business Object Oriented Analysis and Design using BOOAD

We follow the steps below in constructing a model of a system using BOOAD.

- **Identify the outward behavior** of the system by discovering a*ctors* (people or other systems playing a specific *role*) and how each actor interacts with the system. Work at the granularity level of *use cases* [[5]], where a use case is one complete sequence of interactions between the user and the system for the purpose of getting the system to perform a specific function. Each use case will correspond to a system-level operation in the generated product.

- **Specify the user-interface.** The logic of the user-interface of the system will be driven by the use cases discovered above. For each use case, identify precisely the input required from the user (actor), in what order that input will be obtained, and *what is to be done* with that input. User interface logic will be described by the *state transition diagrams* of *User Interface (UI) classes.*

- **Specify the static structure of the system.** Identify *classes, class hierarchies, relationships* among classes (*aggregation* - also called "part-of", *association, inheritance*). Identify *multiplicities.* Differentiate between *user interface classes* and *domain specific classes*. Domain specific classes will in all likelihood be

abstract specifications of databases of the design phase, whereas user interface classes will be used to specify the users' interaction with the system.

- **Specify the dynamic behavior of the system.** Describe the lifecycle of each "significant" object through state transition diagrams. This process will result in the identification of *system events* that will be broadcast/received, as well as the formulation of many queries (in pseudo-code). Differentiate between *system events* and *user interface events*: System events will be used for object communication, whereas user interface events will be generated as a result of an actor interacting with the system.

- **Map user-interface events to system events.** The recipient of user interface events will be user interface classes, which will input data from the user and generate system events with appropriate arguments, after querying classes as necessary to find individual objects to be passed as arguments to the system events (i.e. user interface classes are the links between user interface events and system events). The logic of the user interface class interacting with actors is specified in the state transition diagram of the UI class.

- **Verify the model**. Iterate over the *analysis model*, test it with *scenarios.*

4. Components of a BOOAD Model

In this section we describe each component of BOOAD separately, and in greater detail.

4.1 Classes

In BOOAD, classes are partitioned broadly into three kinds: *domain* classes, *relationship* classes and *user interface* classes. All three kinds of classes can answer *queries* by activating class operations (methods). These queries are formulated using natural language in the analysis phase. Class operations can be invoked directly using the syntax **className::operationName(....)**, or they can be linked to events which cause the operation to be invoked automatically when the event is broadcast. By convention, operations which carry the same signature (name and arguments) as events are activated directly when the event is broadcast.

4.1.1 Domain classes

A *domain* class represents a collection of objects in the problem domain. Domain *classes* (as opposed to instances of a class) do not in general need state transition diagrams (STD) associated with them. They represent collections of objects, and the behavior of the *collection* rarely needs to base its activities on state information. This is not the case for individual objects (i.e. instances of a domain class), which will most likely need STDs of their own. We thus distinguish between a state transition diagram for a class, which represents a set of objects, and a state transition diagram for an individual object, which belongs to a class.

Domain class operations can create instances of themselves, modify and delete objects they contain, and cause the activation of other operations (queries) on other classes.

4.1.2 Relationship classes

Relationship classes are needed to represent *many-many* relationships among objects, or relationships involving three or more objects. They are similar to domain classes in other aspects.

4.1.3 User-interface (UI) classes

UI classes define the user interface of the system. The job of a UI class is to obtain input from the user and generate system events with relevant arguments so that the desired effect will be achieved system-wide through all involved objects responding to the generated event(s). Event arguments are found through appropriate queries to classes.

The behavior of a UI class can be defined very precisely in the form of a state transition diagram. Transitions in the state transition diagram (of a UI class) are triggered by UI events (such as mouse clicks, button pushes, menu choices etc.) generated by the user interacting with the system. Usually, only one instance of the class will be needed, so we tend to regard the class itself as an object and draw the state transition diagram only for the class itself.

4.1.4 Object attributes

Instances of domain or relationship classes can have three kinds of attributes:

- set-valued attributes
- pointer-valued attributes
- simple data attributes

Relationship information about objects are maintained through set valued and pointer-valued attributes: A set-valued attribute contains a set of pointers to objects, and a

pointer valued attribute points to a single object. Simple data attributes, on the other hand, contain data pertaining to the intrinsic characteristics of the object (color, size, etc.).

Notationwise, set-valued attributes start with "~", pointer-valued attributes start with "%" and other attributes of a basic type such as integer, string etc. start with a letter different from "~" or "%".

4.1.5 Operations (methods)

We deal with two kinds of operations (methods) in BOOAD: *instance operations*, and *class operations*. Notationwise, class methods start with "$" whereas instance methods start with any other letter.

4.1.5.1 Instance Operations

Instance operations operate on objects, have access to the internals of the object they operate upon, and are used for changing the state of objects and returning information about the object.

4.1.5.2 Class Operations

Class operations have the responsibility of implementing *queries*, given the role of classes as repositories of their instances. Querying involves adding, deleting, modifying instances of the class, as well as returning a collection of objects satisfying given criteria.

4.2 Events

Communication among objects is an essential component of any OOAD method. The participants of communication include objects, classes and actors. Actors communicate

with user interface classes, and objects and classes communicate with each other. The *event* mechanism is the main communication medium in BOOAD. The other mechanism is *point-to-point*, described in section 5.

4.2.1 Types of events

We differentiate between two kinds of events.

- *User Interface* events are generated when the user interacts with the system, and are detected by user interface classes in their state transition diagrams. These events usually have no arguments (from an analysis point of view), and correspond to a physical action such as a menu item being selected, a button being pushed etc. UI-events are responsible for transitions in the state transition diagram of UI-classes.

- *System* events are generated by user interface classes and other classes or objects with the purpose of carrying information around and causing recipients of the event to take relevant action, possibly using the information carried inside the arguments of the event. System events, together with state transition diagrams, determine the logic of operations in the system.

4.2.2 Communication using events

Communication is initiated when an event is broadcast, with appropriate arguments, by an object or class inside its state transition diagram, or inside a method. The event may be destined for one or more objects and/or classes. Depending upon the current state of the recipient, the event may be accepted, or ignored. If the event is accepted, its arguments are available for use in the accepting entity.

It may be the case that a method of an object or class has the same signature as the event. In such a case, the event is received immediately (irrespective of the current state of the object), and causes the activation of the method with the same signature. This mechanism is explained in detail in section 4.2.3.

4.2.3 Direct activation of methods by events

The mapping between events and operations is achieved implicitly by the state transition diagrams of objects and classes. An exception to this rule occurs when the name of the event coincides with the name, as well as argument types (i.e. *signature*), of a method in a class or an object. In this case, the method in the object/class is invoked immediately regardless of the state the receiving object/class is in. Care should be taken to ensure that methods that are invoked directly on objects do not modify the state of the object. The issue of direct activation of operations in objects is taken up more thoroughly in section 5.

4.2.4 Event parameters

A by-reference semantics is adopted for event arguments in the style of JAVA [[6]] or Smalltalk [[7]]. This means that when an event is received by more than one object, there is only one copy of the objects that are arguments to the event, and these arguments must be shared by the receiving objects. We assume that concurrent access to the methods of the argument objects is controlled, possibly in a similar fashion to the use of *synchronized methods* in JAVA [[6]].

4.3 State Transition Diagrams (STD's)

A state transition diagram, briefly, consists of a set of states, and a set of transitions connecting states. The standard use for state transition diagrams (STD) by OOAD

methods is to describe the lifecycle of significant objects in the system, and consequently the behavior of the whole system from the perspective of individual objects. This is the main use of STD's in BOOAD too. However, they are also used in describing the logic of operations, as well as the logic of the user interaction with the system. This is made possible through allowing transitions to take place without requiring the receipt of an event, and by allowing queries as part of *conditions* and *actions* inside transition. A consequence of these generalizations of STD's is that this gives them the descriptive power of *flowcharts*, and *operations* can be described easily using state transition diagrams.

4.3.1 Transitions

A transition consists of the receipt of an *event* (optional), a *condition* (optional), and the performing of an *action* (optional). Although all three components are optional individually, either a condition or an event must be present in every transition.

A *condition* can involve a query to a class, local variables of the state transition diagram, attributes of the object, a direct message sent to an object found as a result of a query, or a direct message to an object that came as an argument of the received event. The condition part in BOOAD can be made up of more than one condition, each one in square brackets, such as *[condition1] [condition 2]* etc. All the conditions must be true individually before we can proceed to the action part. Furthermore, we can attach *exceptions* to individual conditions to handle errors (that being the reason for separating the conditions from one another) and possibly cause a rollback of the current transaction. The syntax for attaching exceptions to conditions is **[condition]<<exception>>**. An exception is raised if the condition to which it is attached fails to be true. "Built-in" exceptions include **error** (a serious condition which

should not have happened given the semantics of the application), **rollback** (for causing an immediate rollback of the current transaction), and **warning** (for drawing attention to an unusual circumstance)**,** but others may be defined and used for a specific application.

An *action* in a transition can include updating the value of an attribute, invoking an operation of the object, invoking a query on a database, saving results in local variables, and broadcasting event(s).

4.3.2 Local Names in a state transition diagram

We can introduce local names to state diagrams to hold values. Event arguments can also be seen as local variables. The local variables hold their previous values until they are updated by a direct assignment, or the arrival of another event which causes some other value being assigned to an event argument. This is similar somewhat to the dynamic scoping rule of programming languages.

4.3.3 Integrity constraints on the states of a STD

We can state conditions, *integrity constraints*, which must be true in a given state, regardless of what happens. If a transition into a state causes one of the integrity constraints on the state to be violated, an exception, such as **error** or **rollback** can be raised. Integrity constraints are written with the same syntax as conditions annotated with exceptions in a transition.

5. Point-to-point communication in BOOAD

In *point-to-point* communication, *messages* rather than *events* are used. Messages are sent directly to a specific object or class, and activate *methods* in the receiving entity (we adopt the convention that the message name is the same as the method name

which is activated in the entity receiving the message). Here, the identity of the receiver is known to the sender, and both the sender and receiver are unique. No other entities in the system need to be aware of this communication.

Since the identity of the receiver must be known by the sender before a message can be sent directly to the receiver, it may be necessary to do some work find out what this identity is. In the case that the receiver is a class, we can directly use the class name, as in **classX::methodY(…).** If the receiver is an object, and that object's identity is not known to the sender, the object must be found through a query sent to an appropriate class, possibly, but not necessarily, to the class to which the object belongs. An example sequence of actions might look like:

anObject := anObjectClass::"a query";

anObject.someOperation(...).

Note that point-to-point communication by-passes the filtering mechanism provided by state transition diagrams, as in the case where an event automatically activates a method with the same signature. In the state diagram of an object, a method of the object is called only when the object is in a relevant state, and either a condition is true, or an event has been received (or both) which causes the activation of the method. If the directly-called method changes the state of the object to which it belongs, this invalidates the whole state transition diagram mechanism, and the direct call should be avoided. On the other hand, methods that just return information about the object and do not have any side effects can be called at any time without any undesirable effects.

Classes may also participate frequently in point-to-point communication where a class operation is invoked directly by some other entity. This will in general not have undesirable effects, since non-UI classes typically do not have significant behavior that depend upon the current state of the class. UI-classes do have state transition diagrams, but their methods are not meant to be activated directly at all; only when the current state and the received UI-event dictates can one of their methods be activated.

6. Mapping from the Object Model and Dynamic Models of BOOAD to Relational Database Design

Once the analysis and high-level design model for a system has been generated, it is time to map its components onto a *relational database* design. A mapping into object-oriented databases would have been even more straight-forward, but relational database management systems are in such common use today and have matured to such an extent that that we choose to deal with them in this presentation.

There are two main aspects of the BOOAD model that need to be mapped: its *structure*, and its *dynamic behavior*. The static structure is described in the form of classes and their relationship to one another (aggregation, association, inheritance). Dynamic behavior, described implicitly by the state transitions diagrams of objects and classes, is how the system behaves outwardly and what changes take place in the overall system when a certain interaction takes place between an actor and the system.

Before we go on to describe the mapping from the structure and dynamic behavior of a BOOAD model onto a relational design, let us first give a brief introduction to relational database systems.

6.1.1 Relational Databases

In a relational database, information is stored inside *tables*. Tables contain *records*, which in turn contain atomic data inside *fields*. A field has a specific type, such as *character*, *integer*, *string* etc., a name, as well as a (possibly empty) set of constraints which limit the kind of data it can contain. A *database* is a related set of tables and the records contained in the tables.

Before we can start to populate a database with records, the structure of each table in the database needs to be specified using a *data definition language* (DDL). This specification is known as a *database schema*. In a record, a set of fields whose values uniquely identify the object is known as a *key*. Fields forming a key are specified when the database schema is generated.

Populating a database once its structure is specified is done through *queries* specified using a *data manipulation language* (DML). DML is usually very high level and declarative. An example of a DML is *Structured Query Language (SQL)* [[8]] and *Query by Example (QBE)* [[8]]. Logic specific to a database application is implemented using its DML.

6.1.2 Mapping the static structure of a BOOAD model onto a relational database schema

The static structure of a BOOAD model is mapped onto table definitions, with necessary additional information (such as types of fields) being specified. Abstract classes are not directly mapped onto anything in the relational design, but are implicitly used since their attributes, which are eventually inherited by some non-abstract class, become field names in tables.

In a BOOAD model, we assume the existence of object identifiers. In a relational database, object identity is implemented by values contained in the key fields of records, where no two distinct records are allowed to have the exact same data in their key fields.

Relationships of a BOOAD model are mapped quite straightforwardly onto the relational model. *One-one, one-many* and *part-of* relationships depicted in the analysis model can be modeled by one table having a *foreign key* belonging to the other table involved in the relationship (a foreign key is a set of fields that depict a record uniquely in some other table). Many-many relationships between two classes, as well as relationships involving more than two classes necessitate the use of a *relationship table* with foreign keys to the tables involved in the relationship. Relationship classes (when used to depict many-many relationships in the analysis model) are mapped onto relationship tables in a straight-forward manner.

6.1.3 Mapping the dynamic behavior of a BOOAD model onto data manipulation operations in the relational model

Mapping the dynamic behavior of the analysis and high-level design model onto data manipulation operations in the relational world is more involved. In a relational database, data is *passive*, waiting to be manipulated through queries. This is in contrast with the object-oriented approach of the analysis and design model, where objects are assumed to be active. Furthermore, many things seem to be happening concurrently, with each individual object having its own thread of control. In a relational database application, disregarding the fact that more than one user could be interacting with the system at any one time, there is a single thread of control, which possibly modifies the data in a sequential fashion. Thus, the transition needs to be

made from maximum concurrency, to an individual thread of control, and there should be no loss in the semantics of what the system does outwardly.

We start by identifying the system-wide functions which are initiated by the user's interaction with the system. It is the job of a UI class to guide the interaction, get relevant data from the user, make queries on the classes, and generate events which will be picked up by individual objects and/or classes. Upon the receipt of an event, objects may change state, make queries, etc. In the relational model, records, which correspond to the objects receiving messages, are passive. Hence, any state changes to objects need to be performed on their behalf. Since possibly there is more than one recipient of an event, the same thing needs to be done for each recipient. Furthermore, any other actions taken in the state transition diagrams of objects upon the receipt of an event need to be performed on their behalf. If these actions include the broadcasting of other events, these events must be traced (recursively) and the state changes they cause implemented for the receiving objects. In fact, there can be a chain of events being generated, and each one can cause state changes in individual objects, and hence the overall system. We should trace the broadcasting and reception of all such events, and make sure that data manipulation operations perform the required state changes on behalf of the records.

Class methods are another aspect of the dynamic behavior of the system. These can be mapped quite straightforwardly to queries using the DML of the relational database management system (RDBMS).

7. Sample Analysis model: Student Course Registration

In this section, we demonstrate the workings of BOOAD using a very simplified version of the student registration procedures in a university. To keep the presentation reasonably small, we left out the user-interface classes. We thus assume that for each transaction, an appropriate user interface class reads data from the user, and generates system events which will be picked up by other classes and objects in the system.

7.1 Static Analysis

The (invisible) actors of the system are the registrar's office personnel. They process data from the other active elements, i.e. students and teachers, and feed the system with that data.

Objects which participate in the system have classes representing them. These classes are **Person**, an abstract superclass from which classes **Student** and **FacultyMember** are derived, **CourseOpening** which represents a specific course section opened in a semester, **Course** which gives the description of a course and **StudentTakesCourse** which is needed to record the many-many relationship between **CourseOpening** and **Student** classes. The static description of the system is given in Figure 1.

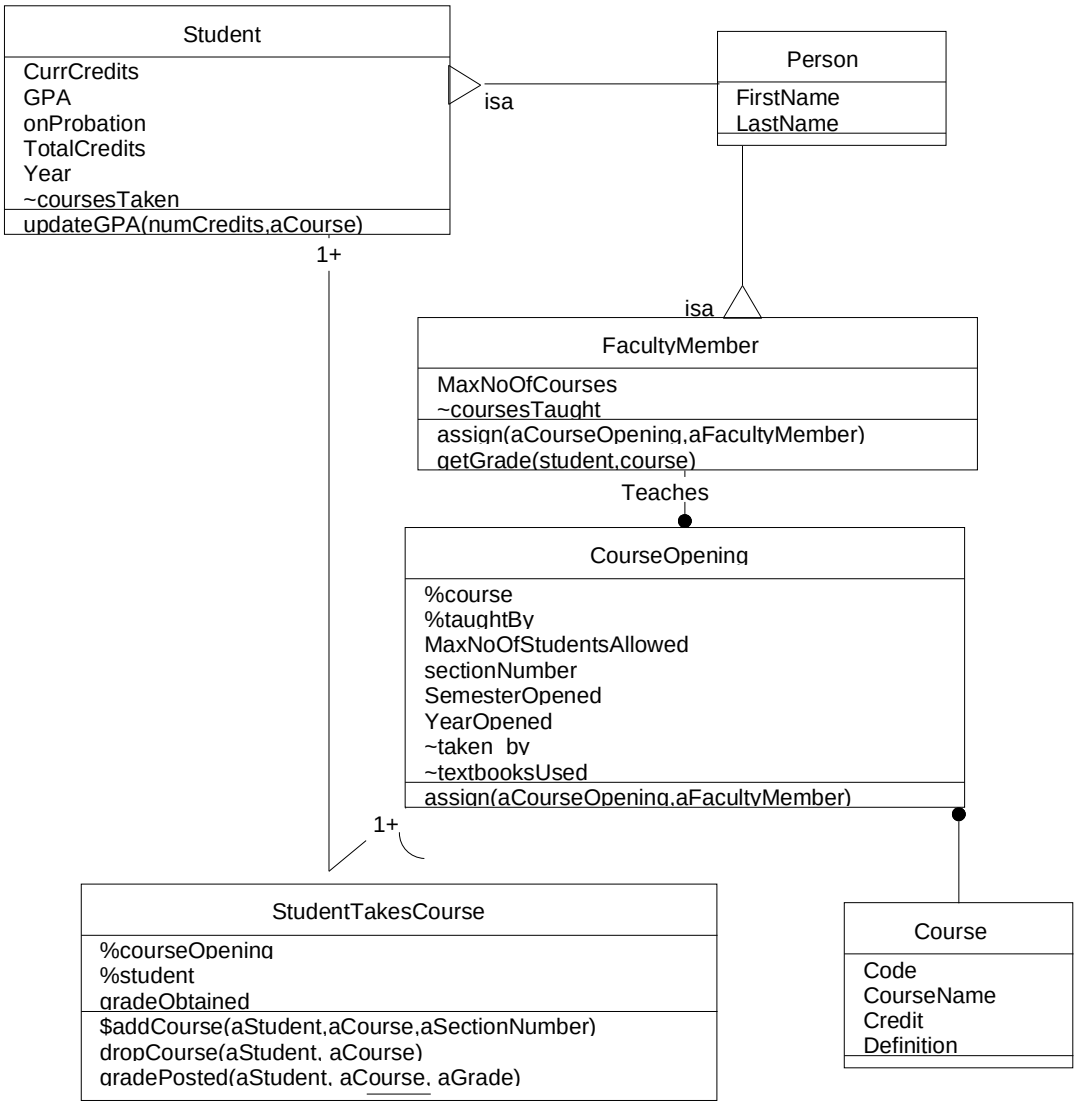

Figure 1: Static model of the course registration system

7.1.1 Person

Person has properties common to students and faculty members, i.e. *firstName* and *lastName*.

7.1.2 Student

Student extends **Person** and adds the fields *currCredits* (the total number of credits being taken in the current semester), *GPA* (grade point average at the beginning of the

semester), *onProbation* (whether the student is currently on probation or not, decided according his GPA), *totalCredits* (the total amount of credits he had at the beginning of the semester), *year* (one of "Freshman", "Sophomore", "Junior", "Senior"), and *~coursesTaken* (set of pointers to **StudentTakesCourse** objects). The operation *updateGPA(numCredits,aCourse)* updates the GPA of the student (at the end of a semester), as well as his total credits based upon his status and grade obtained in aCourse.

7.1.3 FacultyMember

FacultyMember extends **Person** and adds the fields *maxNoOfCourses* (the total number of courses the faculty member can be asked to teach), as well as *~coursesTaught***,** a set valued attribute containing pointers to **CourseOpening** objects. The operation *getGrade(student,course)* returns the grade the first argument (student) obtained in the course taught by the faculty member. *assign(aCourseOpening, aFacultyMember)* is both an operation of **facultyMember** and an event generated by some UI class. As such, it is automatically activated when the event is broadcast. If the faculty member is the intended recipient, then the object **aCourseOpening** is added to the *~coursesTaught* set.

7.1.4 CourseOpening

Instances of **CourseOpening** represent different sections of a course opened in a semester. Its fields include *%course* (pointer to a **course** object of which this instance is a section), *%taughtBy* (pointer to the faculty member teaching the course), *MaxNoOfStudentsAllowed* (to take the course), *sectionNumber*, *semesterOpened* ("Fall", "Spring", "Summer"), *yearOpened* and *~takenBy* (set valued attribute containing pointers to **studentTakesCourse** objects). *assign(aCourseOpening,*

aFacultyMember) has a similar function to that in **facultyMember** and result in the setting *~taughtBy* to point to **aFacultyMember.**

7.1.5 StudentTakesCourse

Instances of **StudentTakesCourse** are used to implement the many-many relationship between **courseOpening** objects and **Student** objects. *%courseOpening* thus points to a **courseOpening** object and *%student* points to a student object. *gradeObtained* (the grade the student obtained by taking the course implied by *%courseOpening*) is conveniently located in instances of **StudentTakesCourse.** The class operation *$addCourse(aStudent,aCourse,aSectionNumber)* creates an instance of **StudentTakesCourse** and initializes its fields after making appropriate queries. **dropCourse(aStudent, aCourse)** is an instance method, which sets *gradeObtained* to "dropped." *gradePosted(aStudent, aCourse, aGrade)* is similar, but sets *gradeObtained* to **aGrade.** All these operations are automatically activated upon the receipt of events by the same signature.

7.1.6 Course

Course objects represent course definitions (somewhat like catalog data), independently of when the course is opened, who is teaching it etc. Fields of a **course** object include *code* (such as CmpE 420), *courseName* (name of the course spelled out, e.g. "Principles of Programming Languages"), *Credit* (how many credits the course is worth), and *Definition* (a paragraph of information about the contents of the course).

7.2 Dynamic Analysis

Dynamic analysis reveals the behavior of individual objects in the system from their own point of view. This is accomplished through state transition diagrams, or by direct activation of methods inside objects and classes by events.

7.2.1 Dynamic Behavior of a faculty member

During registration, courses are assigned to faculty members. The number of courses assigned should not exceed the total number of courses he is supposed to teach (this may vary according to rank, other duties etc.). After semester has ended, a faculty member posts grades for students who have taken his courses.

The state diagram for a faculty member is given in Figure 2. Note that a query is sent to "system." This is done when there is no obvious class that should answer the query.

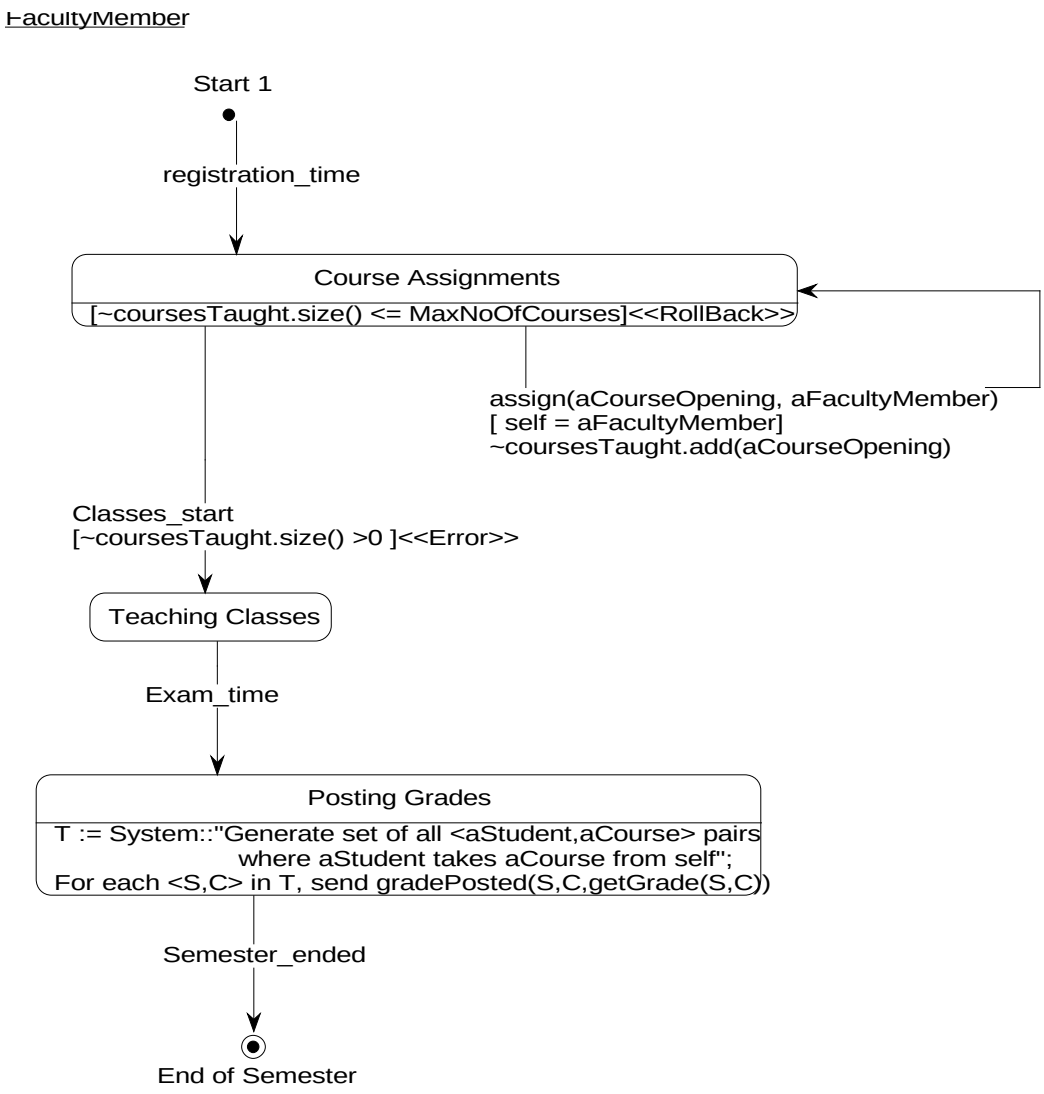

Figure 2: State transition diagram for a facultyMember object

7.2.2 Dynamic behavior of a student

The status of a student is determined at the start of a semester. If his GPA is greater or equal to 2.0 then he can register as a regular student. Otherwise he is "on probation" and can only repeat courses that were previously taken with a grade lower than "C." A regular student can take courses whose total credits do not exceed 20 and are not less than 15 either. There is no minimum credit requirement for students on probation. Regular students can drop courses during the semester, provided their load does not

fall below 15. Students cannot add courses during the semester. Once the semester is over and grades are posted, students update their GPA and total credits taken.

Figure 3 depicts the state transition diagram for a student. Note the widespread use of conditions with special actions attached to them (to be activated if the condition is **not** true). Also of interest is the use of events generated externally which affect the system, such as *Exam_time* and *Classes_Start*, and the use of 'C'-like operators for brevity, such as x+=y, meaning x:=x+y etc.

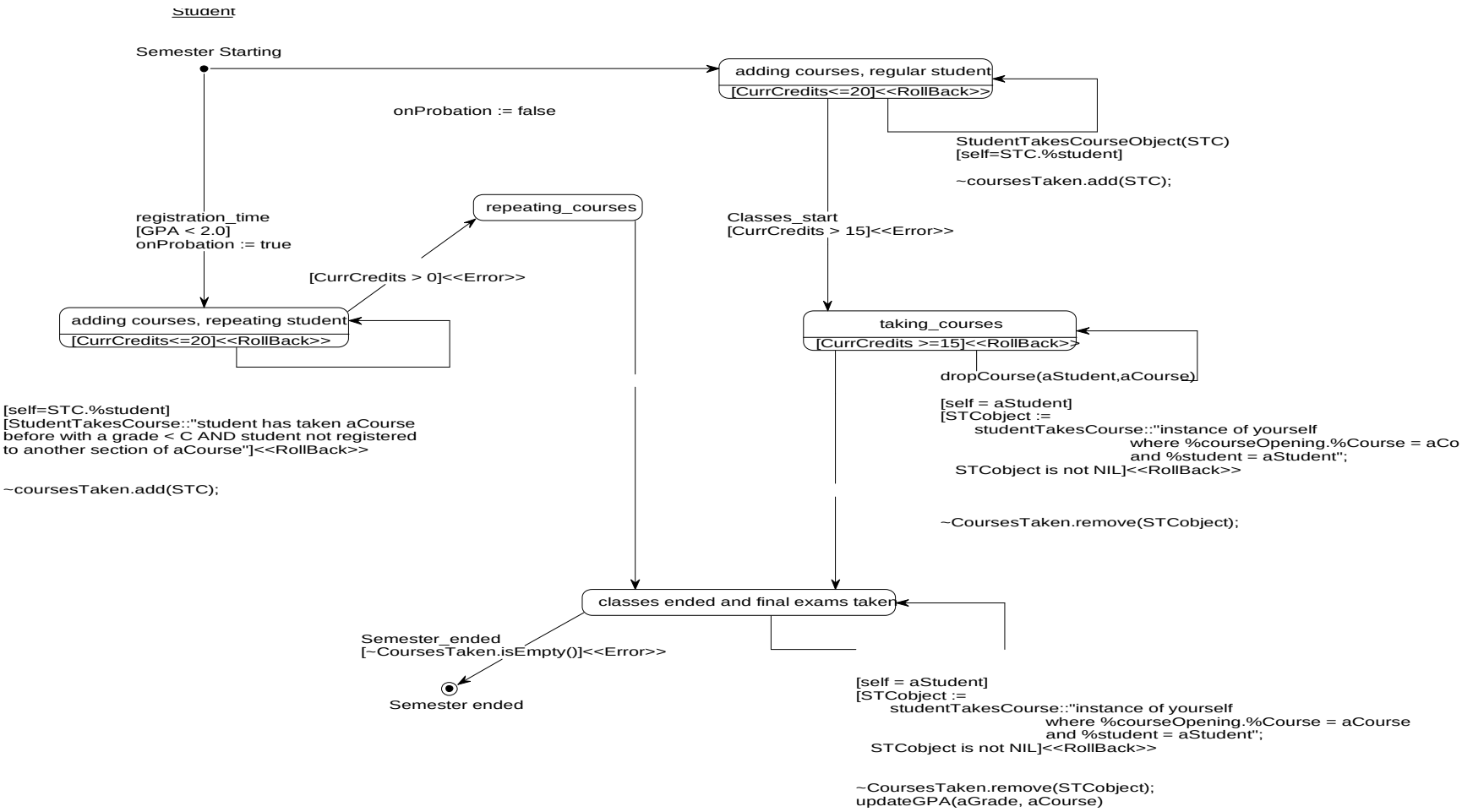

Figure 3: State transition diagram for a student object

7.2.3 Dynamic behavior of a CourseOpening object

A **courseOpening** object is created for each section of a course opened in a semester.

The state diagram of this kind of object is given in Figure 4.

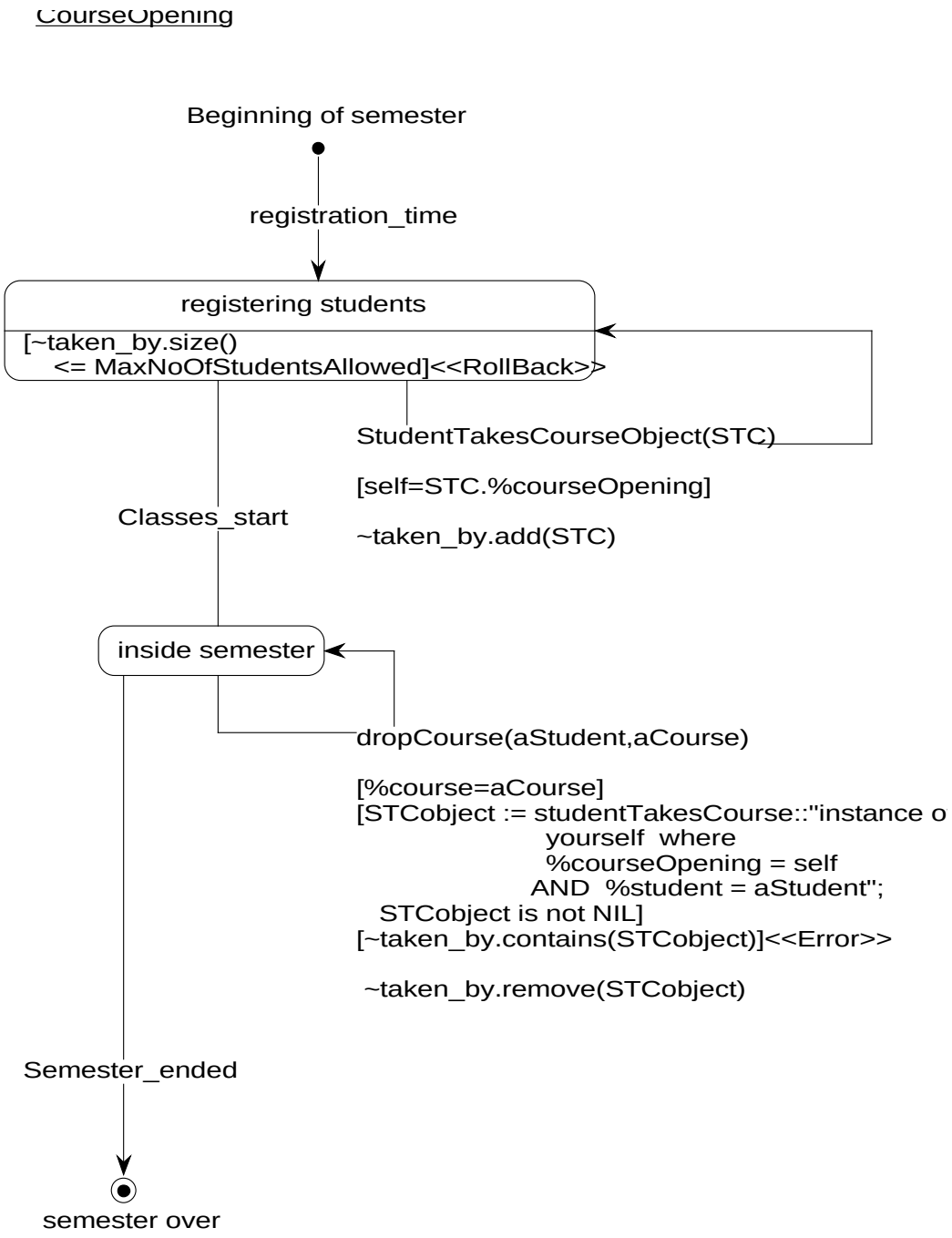

Figure 4: State transition diagram for a courseOpening object

7.2.4 Dynamic behavior of StudentTakesCourse objects

Objects of class **StudentTakesCourse** are used to implement the many-many relationship among students and **courseOpening** objects. Their dynamic behavior is simple enough for them not to require state transition diagrams: class and instance methods suffice. We give below these operations.

The first of these, *$addCourse(aStudent,aCourse,aSectionNumber),* given in Figure 5, is a class method which has the same name as an event, and is thus activated automatically as soon as the event is broadcast (by a UI state transition diagram). First, a check is made to see whether the student given in the argument has taken the course before with a grade "C" or better, or if the student is registered for a different section of the same course. This is done through a query to the class (self here refers to the class, since *$addCourse(aStudent,aCourse,aSectionNumber)* is a class method). If that is the case, the transaction is rolled back. Otherwise, a new instance of **studentTakesCourse** object is created, and the links of this object is set to point to **aStudent** and to the appropriate **courseOpening** object, which is found after a query to the **courseOpening** class. Finally, a one-argument event, *studentTakesCourseObject(STC),* is generated, the argument being the **studentTakesCourse** object just created. This event will be received by **student** and **courseOpening** objects.

```
$addCourse(aStudent,aCourse,aSectionNumber)
IF self::"aStudent has  taken
          aCourse before with a grade >= C or is
          currently registered for a different section of
          aCourse"
THEN <<ROLLBACK>>
ELSE
  STC := self.newInstance();
  STC.%courseOpening:=
    CourseOpening::"an instance Y of yourself
                    such that Y.%Course= aCourse and
                    Y.sectionNumber = aSectionNumber";
   STC.%student := aStudent;
   send StudentTakesCourseObject(STC);
ENDIF
```

Figure 5: Class method addCourse of studentTakesCourse

Next we have the instance method *dropCourse* in Figure 6. This method has the same name as an event, and is activated automatically when the event is broadcast. Every **studentTakesCourse** object will then be executing this method when the event is broadcast. However, only the one with links to the correct **student** and **courseOpening** objects will do further processing, i.e. setting GradeObtained to "Dropped."

```
dropCourse(aStudent, aCourse)
If %student =  aStudent
  AND  %courseOpening.%course = aCourse
  THEN
  GradeObtained :=  "Dropped"
```

Figure 6: Method dropCourse of studentTakesCourse objects

Finally we have the instance method *gradePosted(aStudent, aCourse, aGrade)*, which is similar to *dropCourse(aStudent, aCourse)* and updates the *gradeObtained* field of the object.

```
gradePosted(aStudent, aCourse, aGrade)
IF  %student =  aStudent AND
    %courseOpening.%course = aCourse
THEN gradeObtained := aGrade
```

Figure 7: Method gradePosted of studentTakesCourse objects

8. Conclusion and further work

We described an object-oriented analysis and design methodology, BOOAD, as an example of an object-oriented *application-area and computation-paradigm specific* analysis and design methodology tailored for database applications. Its notation and semantics is inherited from OMT, but enriched with notions lifted up from object-oriented database systems. These lifted-up notions include transactions, queries, integrity constraints, set-valued and pointer-valued attributes. Other enhancements include an exception mechanism for dealing with errors.

The advantages of the presented approach include the minimization of the potential difficulties in making the transition from analysis to design and implementation. Furthermore, we have a semantically rich set of notations and concepts available at the analysis phase which are of great use in describing systems with database functionality, as was demonstrated in the student registration example.

We expect other *application-area and computation-paradigm specific* methodologies to emerge in time for other application areas and computation paradigms. We envision analysis-design methodologies for numeric-intensive programs running on parallel architectures, computer games with involved user-interface components, embedded real-time systems, robot controllers, secure systems where certain risks need to be minimized, verifiably correct systems and the like.

Further work on BOOAD includes the implementation of tools to support the proposed methodology.

Acknowledgments

The author would like to thank Nahit Emanet for his useful comments on an earlier draft of this paper.

References

[1] James Rumbaugh, *OMT Insights: Perspectives on Modeling from the Journal of Object-Oriented Programming*, Prentice Hall, 1996

[2] Grady Booch, *Object-Oriented Analysis and Design with Applications*, Second Edition, Addison-Wesley 1994

[3] Derek Coleman, Patrick Arnold, et al., *Object-Oriented Development: The Fusion Method*, Prentice Hall, 1994

[4] Michael Blaha and William Premerlani, *Object Oriented Modeling and Design for Database Applications using OMT and UML*, Prentice Hall, 1998

[5] Ivar Jacobson, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, 1992

[6] Ivor Horton, *Beginning Java*, Wrox Press, 1997

[7] Simon Lewis, *The Art and Science of SmallTalk*, Hewlett Packard, 1995

[8] Jeffrey D. Ullman, Principles of Database and Knowledge-Base Systems, Vol. 1, Computer Science Press, 1988