An Object-oriented Systems Modelling Method based on the Jackson Approach

DANNY C. C. POO AND SHWU-YI LEE

Department of Information Systems and Computer Science, National University of Singapore, Kent Ridge Road, Singapore 0511

This paper proposes an object-oriented information systems development method that is based on the modelling approach of the Jackson method. Like the Jackson method, the proposed method separates the consideration of the real world subject matter about which a system is to compute from the function to be provided by a system for its users. Events and functions are the main information sources by which domain objects and other function-related objects are identified and modelled. The modelling approach of the proposed method is thoroughly discussed with the use of a Student-Records System as the mainline example.

Received June 10, 1994, revised September 15, 1994

1. INTRODUCTION

The lack of well-defined management strategy and the piecemeal development approach of traditional systems development methods had posed a number of problems for practitioners. Martin (1983) noted the following shortcomings inherent in the traditional approach:

- The ambiguities inherent in narrative descriptions have led to poor requirements specification of systems.
- The lack of rules or heuristics to assist developers in transforming a requirements specification into a design specification results in poor designs.
- There are difficulties in system maintenance.
- Poorly designed systems results in more time and effort spent in maintaining systems. Consequently, less resources can be devoted to new applications leading to backlog.

The response to these problems has been the introduction of methods for constructing software systems based on a system development life cycle framework. While methods provide a systematic approach to developing software systems, they do not necessarily lead to the production of maintainable systems. The derived systems may not necessarily have provisions for accommodating future changes. Software engineering practices that promote the localization of effects of change through principles like encapsulation, loose coupling and high cohesion have been seen as essential in design models by most software practitioners (e.g. Meyer, 1988; Booch, 1991; Jacobson, 1992). Software development methods must therefore provide developers with not only a systematic step-by-step approach to producing systems but also enable the construction of maintainable systems right from the start in the modelling process. To this end, the object-oriented design technique has made significant contributions. Many object-oriented methods have been proposed over the years (e.g. Meyer, 1988; Booch, 1991; Wirfs-Brock and Wilkerson, 1989; Coad, 1992; Jacobson, 1992; Martin and Odell, 1992). These methods provide notations and guidelines for developers to create and implement a model of the application domain. These methods generally address the following concerns:

1. *How to create an object model.* Most models recommend the use of nouns as the starting point in object model construction. The Entity-Relationship modelling technique is generally used as the base technique for structuring the object model. Other modelling tools such as data flow diagrams and state transition diagrams supplement the core object model. The problem with this approach is the difficulty in sieving out inappropriate objects when the list is large. Inter-relating various models that do not have a direct connection with one another is another problem that these methods suffer.
2. *Software maintenance.* Most of these methods address the development approach very well. However, a system produced using these methods would only achieve software maintainability from the advantages attributed to the strict adherence of software engineering principles such as encapsulation and loose coupling between parts advocated in the object-oriented paradigm. Functional requirement changes, often seen as the major contribution to changes in a system, are not well provided for in these methods.

The development approach advocated by the Jackson method (see Jackson, 1983; Layzell and Davis, 1993) separates the consideration of real world subject matter about which a system is to compute from the functions to be provided by a system for its users. The advantage of this approach is the ability to separate the volatile functional requirements of a system from the stable subject matter that defines a system domain. In this way, the effect of change could be localized within the respective functions when changes in functional requirements arise.

A high level of system maintainability can thus be achieved. However, since the Jackson method is not an object-oriented method, much of the advantages available in the object-oriented approach are not supported.

In this paper, we discuss an object-oriented software development method that aims to address the above two concerns. The development approach of the proposed method is inspired by the approach advocated by the Jackson method (Jackson, 1983). As Layzell (Layzell and Davis, 1993, p. 14) pointed out 'the most relevant aspect of real world subject matter is when something happens to it (or the occurrence of an event)'. The proposed modelling process thus begins with the identification of events. From events, a set of objects known as the domain objects are identified. Functions to be provided by a system are separately considered. With these functions, other types of objects that closely model the requirements of functions are also identified and added onto the initial model created through the events. The final framework of the resultant system model is thus made up of a set of interacting objects.

The following sections of this paper will discuss the proposed object-oriented method. They are organized in the following manner:

- Section 2: overview of the proposed method.
- Section 3: event modelling.
- Section 4: function modelling.
- Section 5: consolidation of objects identified through events and functions.
- Section 6: dynamic behaviour specification through use scenarios.
- Section 7: Conclusion.

Like other methods, the proposed method is not concerned with how user requirements are established.

2. THE MODELLING APPROACH

The proposed development method has five steps. In step 1, a list of events and functions is drawn out with the help of an augmented context diagram. In step 2, an event script is described for each event identified in step 1. Two types of objects are identified in this step, i.e. domain objects and supporting objects. Further objects are identified by considering the functions that a system is required to support. This is done in step 3 through function scripting. Objects and their relationships are incrementally identified through the first three steps. Step 4 is a consolidation step. It examines the objects identified in the previous steps and produces a consolidated set of objects that would appropriately represent the domain under investigation. Step 5 considers how a system will be used and, through the use of use scenarios, it defines the interactions among objects.

3. EVENTS AND FUNCTIONS

The method begins with the construction of an augmented context diagram. An augmented context diagram is used to illustrate the source of events in terms of event triggers. Functional requests representing the functional requirements that a system has to fulfil for the users are also included in the diagram. Event triggers and functional requests reflect the happenings in the real world (see also Layzell and Davis, 1993). By modelling event triggers and functional requests in an augmented context diagram, we aim to derive and express the responsibilities of a system.

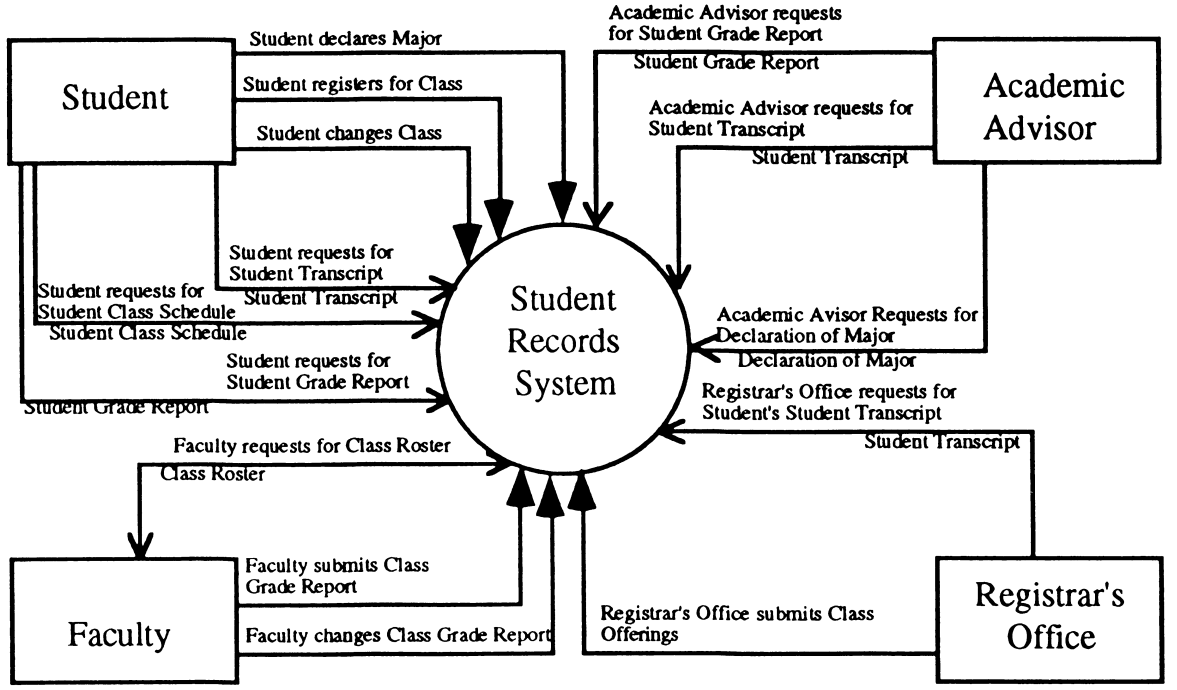

FIGURE 1. An augmented context diagram for a Student-Records System.

Figure 1 illustrates an augmented context diagram for a Student-Records System; it shows:

- The event triggers from its various sources such as Student, Faculty and Academic Advisor. This is indicated by an arrow with a darkened arrow head.
- The functional requests on a system such as Student requests for Student Transcript. This is indicated in the diagram by a regular arrow.
- The data output generated as a result of the functional requests, e.g. Student Transcript, Student Grade Report and Class Roster. This is indicated by the label below a functional request arrow.

For each event trigger reflected in the diagram, an event will be described. An event is an instantaneous activity that happens in the real world. An event causes a change in the state of a system which describes how far the system has come into execution. For example, an event in a Student-Records System might be *student register for class*. The source of this event is Student.

Fulfilling a functional request does not change the state of a system. An example of a functional request in a Student-Records System is *student requests for student transcript*.

From Figure 1, we note that the Student-Records System has the following events:

- Student registers for class.
- Student declares major.
- Student changes class.
- Faculty submits class grade report.
- Faculty changes class grade report.
- Registrar's office submits class offerings.

The identified functional requests and their corresponding data outputs are:

- Student requests for student transcript, student transcript.
- Student requests for student grade report, student grade report.
- Student requests for student class schedule, student class schedule.
- Faculty requests for class roster, class roster.
- Academic advisor requests for student transcript, student transcript.
- Academic advisor requests for student grade report, student grade report.
- Academic advisor requests for declaration of major, declaration of major.
- Registrar's office requests for student transcript, student transcript.

The augmented context diagram forms the building

Event S.2e Student Registers for Class

Data: student number, term id, class number, section number
From: STUDENT
Temporal Info: Beginning of term

Meaning:
STUDENT may register for a certain number of CLASSES, specifying the SECTION of the CLASS he wishes to register for, provided:
i) the STUDENT meets the PREREQUISITES for the CLASS (if any)
ii) there are seats available for that CLASS
—if the SECTION that he wish to register for is filled, i.e. seats available = 0, alternate SECTIONS of the CLASS may be offered to him for selection (if there is any)
iii) there is no conflict with the requested SCHEDULE

Activity:
1. Verify the existence of STUDENT using student number
2. For the CLASS that STUDENT wants to register for:-
   i) verify the existence of CLASS using class number, term id
   ii) verify the existence of SECTION using section number
   iii) check for existence of PREREQUISITES for the CLASS
      — if PREREQUISITES exists, verify that the STUDENT meets the PREREQUISITES
   iv) check if there are seats available for the SECTION of the CLASS
      — if there are no seats available for the SECTION, check if there are alternative SECTIONS of the CLASS that has seats available and do not have conflicts with the SCHEDULE of STUDENT
         — if there are alternative SECTIONS for the CLASS, offer these SECTIONS for selection
      — verify that there are no conflicts with the SCHEDULE of STUDENT
         — if there is conflicts, check if there are alternative SECTIONS for the CLASS with seats available and do not have conflicts with the SCHEDULE of STUDENT
            — if there are alternative SECTIONS for the CLASS, offer these SECTIONS for selection
3. For the CLASS that STUDENT can register
   i) STUDENT is registered for the CLASS, i.e. the seats available for the SECTION of the CLASS that the STUDENT is registered for is decrement by 1
   ii) (class number, section number, days-offered, time-offered, location) is added to SCHEDULE

Post-Event Activity:
Output the Student Class Schedule.

FIGURE 2. Event script of Student Registers for Class.

block for in-depth analysis of the application domain. From the list of event triggers and functional requests, we could identify an initial set of events and functional services that a system has to support. Further events and functional services may be identified at a later stage of the analysis.

3.1. Identifying domain objects through event scripting

For each event trigger in the context diagram, an event script is described. From the event script, we aim to identify the relevant objects connected with the problem domain.

Figure 2 shows an example of an event script of event Student Registers for Class. An event script contains the following information:

- The event trigger.
- Event data.
- The source of the trigger and relevant data connected with the event.
- The temporal information on the event.
- The event activity.
- The post-event activities.

An event may be linked to another event and this is described as a 'post-event activity'. A post-event activity may perform further functions or trigger the execution of another event. An example of a post-event function is the generation of a Student Class Schedule. By including post-event activity, the event-scripting process enables the analysts to determine the relationships among the scripts; it also helps the analysts to identify additional functions or events that may not be included in the context diagram initially. Through this process of drawing context diagram and event scripting, we incrementally build up the system model.

An event script provides the first source of information for identifying objects. For example, from the event script, Student Registers for Class, we identify Student and Class as objects since they are participants of the event. A participant of an event is one which causes a change in the state of a system. This type of objects is known as *domain objects*. For each domain object that participates in the event, we define an operation that corresponds to the object's activity in the event. This operation is known as an *action*. An action is an operation of a domain object that causes a change in the state of the object. Hence, there will be an action for Student and Class corresponding to the event Student Registers for Class. For simplicity, we shall call this action, Register. Thus, Student and Class will each have a Register action.

In addition to domain objects, there are also other objects that are involved in an event. These objects play a supportive role in the event. Their presence in the event does not change their own state nor the state of the system as a whole. They are specified and defined for the purpose of the event. This type of objects is known as *supporting objects*. An example of a supporting object in the event Student Registers for Class is Schedule. This object keeps information of a student's schedule. The operations performed by Schedule do not contribute to any change in the state of the Student-Records System.

There are two main differences between a supporting object and a domain object:

1. Supporting objects do not have actions since their operations do not change the state of a system.
2. The state of supporting objects is insignificant. In other words, we are not concerned with the state a supporting object can take.

Supporting objects are distinguished from domain objects to provide analysts with a level of abstraction for identifying objects.

3.2. Identifying object relationships

Objects are related to one another and events form a basis for deriving the relationships among objects. For each event script, we construct an object-relationship diagram indicating the relationships among the domain and supporting objects in that event. An object-relationship diagram is a graphical representation of the static links that exist amongst objects in an application domain. Constructing the object-relationship diagram at event level allows the analysts to focus on a smaller subset of objects that exist in an application domain.

Figure 3 illustrates the relationships among the objects in the event Student Registers for Class. The relationship between Student and Class is an association; this relationship is represented by a line connecting the two objects. The association between Student and Class is derived from the fact that both objects participate in the same event Student Registers for Class. These two objects are said to possess common actions (Jackson, 1983; Layzell and Davis, 1993). A relationship has cardinality constraint represented by the vertical bars on the line connecting objects. A single bar represents one

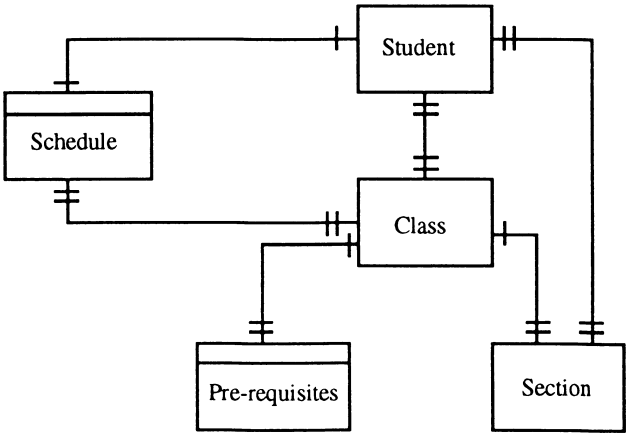

FIGURE 3. Object-relationship diagram for the event Student Registers for Class.

instance relationship, a double bar represents a many (one or more) instance relationship. Since a student can register for many classes, the relationship is a many instance relationship; therefore, a double bar is used on the link closer to the Class object. Similarly, for each class, there can be one or more students registered for the class, hence, the relationship is a many instance relationship. Thus a double bar is also used on the link closer to the Student object. Supporting objects are distinguished from domain objects by an extra broken line at the top of the box.

3.3. Identifying object attributes

Object attributes are also derived from event. Consider again the event, Student Registers for Class. This event requires student number, major code and minor code which are information associated with the Student object. These information thus represents the data attributes of the Student object. Student number is included to identify the specific instance of Student. The major code and/or minor code are included for the purpose of checking on the prerequisites which is connected with the event. For each association, we identify link attributes to connect the objects in the relationship. A link attribute is the key that uniquely identifies the other object. In this example, the link attributes for Student is depth. id, class number, term id. These link attributes will be used to identify the class that a student is registered with. Figure 4 shows the definition of the domain object, Student, based on the event Student Registers for Class.

3.4. Identifying object services

Besides actions, a domain object has other operations; these operations do not update the state of the object and are known as *services*. Services can be distinguished as class services or instance services. A class service is an operation that belongs to a class but not the instances of

STUDENT

Object Type: Domain Object

Attributes:
student number
major code
[minor code]

Link Attributes:
{ dept. id
class number
term id
}*

Constraint Policies:
Maximum no. of courses per term = 8

Event: Student Registers for Class

Action:
Register(class number, term id, section number)

Class Service:
StudentExists(student number)
—return Boolean value

Service:

FIGURE 4. Composition of Student for event Student Registers for Class.

SCHEDULE

Object Type: Supporting Object

Attributes:
registration date
{ class number
section number
days offered
time offered
location
}*

Link Attributes:
student number
class number

Event: Student Registers for Class

Class Service:
ScheduleExists(student number, class number)
—return Boolean value

Service:
conflict(days-offered, time-offered)
—return Boolean value
addcourse(class number, section number, days-offered, time-offered location)
—no return values

FIGURE 5. Composition of supporting object, Schedule.

the class. Generally, a class service defines an operation that acts on instances of the class. An instance service, on the other hand, is an operation that is applicable only to the instance. For the event Student Registers for Class, we identify StudentExists(student number) as a class service of Student. This service determines if a student object having a particular student number exists in the system. An instance service for a Student object might be GetName() or GetAddress(). These services will return the name and address of a particular student respectively.

Therefore, an operation in the proposed approach is distinguished as an action, a class service or an instance service.

The definitional description of supporting objects is similar to that of domain objects except that a supporting object does not have actions. The definition process proceeds by examining the static information that a supporting object contributes to the execution of an event. In the Student Registers for Class event, we note that the supporting object Schedule has to hold information pertaining to the time-table of the Student object. The attributes of the Schedule object will therefore include class number, section number, days-offered, time-offered and location. Figure 5 defines the Schedule object. Since a Schedule object is related to the domain objects Student and Class, it will thus have student number and class number as its link attributes. Identification of the class services and instance services for supporting objects is also based on the operations needed to facilitate and support the activities of the event.

4. FUNCTION SCRIPTING

A function script describes a function due to a functional request as indicated in the context diagram. It is an activity in the system that yields data values from the system. Examples of functions are producing a weekly report, producing a student transcript, printing a Schedule, etc. Functions do not change the state of a system.

Function S.1f Student Requests for Student Grade Report

Input: student number
From: Student
Temporal Info: At the end of term, after the Academics have submitted the class grades for all the classes

Meaning:
The Student Grade Report of a student can be printed out at the end of the term, after the academics have submitted the class grades for all the classes.

Activity:
1. Create Student Grade Report
2. Verify the existence of STUDENT using student number
3. Get (student number, student_name, address, phone number) of STUDENT
4. For every CLASS that STUDENT is registered for during the term using term id
   — get the (class number, course name, credit hours, student grade)
   Add credit hours to total hours
5. Derive the hours-attempted, the hours-completed, the credits-earned, the term-GPA, the cumulative-hours-attempted, the cumulative-hours-completed, the cumulative-credits-earned, cumulative-GPA

FIGURE 6. Function script of Student Requests for Student Grade Report.

A function can be activated from a source as indicated in the context diagram or as a post-event activity.

The purpose of function scripting is to help analysts identify further objects and to refine the definition of domain and supporting objects identified earlier through event scripting. Refinement of definition is carried out by examining the requirements of functional requests and making changes to the already identified domain and supporting objects. These changes include adding further data attributes or services to satisfy the requirements.

Figure 6 is a function script corresponding to the functional request Student Requests for Student Grade Report (see Figure 1). The structure of a function script is similar to an event script except that there is no post-function activity for a function. A function does not activate another function or any other events since its purpose is to yield information from a system. A function script includes the following information: the input data required, the source of the data, the pre-conditions for the successful execution of the function and the description of the function.

4.1. Agent objects

In the assignment of responsibility for the management of functions, a function that pertains to one object type is assigned to that object type. For example, the functional request student's declaration of major. The output data flow connected with this functional request is made up of information such as student number, student name, address, phone number, major code, minor code and registration date. This request is directly connected with an object of type Student. Since this function can be satisfied directly in Student, we shall specify it within the definition of Student. We do so by defining a service as part of the Student's set of operations.

However, for a functional request that involves more than one object type, another object possibly of a different type is introduced to handle the execution of the function. An example will illustrate this. Let us consider the functional request Student Requests for Student Grade Report. This request is connected with two types of objects, Student and Class. For this case, a new object, Student Grade Report is introduced. This object will manage the control flow in the function. We call this type of object an *agent object*. An agent object, as the name implies, is an object that renders services on behalf of other objects. It acts as a controller responsible for the collection of information. For example, the agent object, Student Grade Report, is responsible for the collection of information pertaining to the class grades of a student. It also computes information such as hours attempted, hours completed, etc., and generates these information in a format required by the user. To achieve these, the Student Grade Report agent object has to interact with the Student object as well as the Class object.

From a design point of view, the use of agent objects encourages localization of effects of changes. To illustrate, let us consider the agent object Student Grade Report. This object would handle the collection of information from the various objects, i.e. Student and Class objects to fulfil the functional request. Since Student Grade Report contains the format information of the report, any future changes to the format would be localised within the agent object Student Grade Report. The other two objects, Student and Class objects, would not be affected by the changes. However, if the responsibility of producing the Student Grade Report is assigned to either the Student or Class object, any format changes to the report would require making changes to the Student or Class object. One side effect of this arrangement is the need to determine which of the two objects should the responsibility be delegated to. By using the agent object, Student Grade Report, the delegation of the responsibility is clear and unambiguous. This would thus enhance the software maintenance process (see also Jacobson, 1992).

4.2. Determine object relationships

The objects identified in function scripting are also inter-related to one another. We use an object-relationship

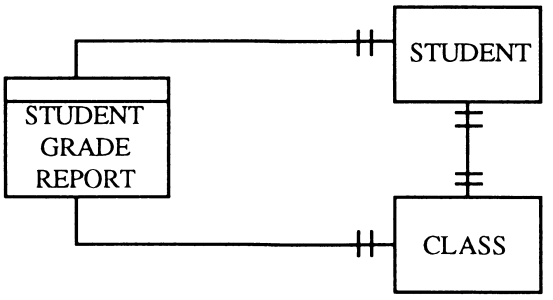

FIGURE 7. Object-relationship diagram for the function Student requests for Student Grade Report.

diagram to depict such relationships. Figure 7 is an object-relationship diagram for the objects identified for the function Student Requests for Student Grade Report. An agent object is indicated in the diagram as a box with a continuous line at the top of the box. Figure 7 shows that the agent object, Student Grade Report, is linked to Student and Class objects. The relationships between an agent object and the other objects are uni-directional and this is reflected by the cardinality constraint on the relationships in the object-relationship diagram. In Figure 7, we note that Student Grade Report is related to Student and Class. However, we are not interested if Student and Class is related to Student Grade Report.

The definition of an agent object includes:

1. Data attributes to facilitate the processing of the corresponding function.
2. Class services.
3. Instance services required by the function.

Figure 8 shows the composition of the agent object, Student Grade Report, for the corresponding function Student Requests for Student Grade Report. An agent object does not have actions since the state of an agent object is insignificant too. Changes to an agent object's data attributes are made for administrative purpose only.

Student Grade Report

Object Type: Agent Object

Attributes:
term id
{ grade
credit hours
}*
total hours
total hours attempted
total hours completed
credits earned
term-GPA
cumulative hours attempted
cumulative hours completed
cumulative credits-earned
cumulative GPA

Link Attributes:
student number
{ class number }*

Function: Student Requests for Student Grade Report

Class Service:
Create(student number)
—no return values

Service:
computeGPA()
—no return values
computeHrsAttempted()
—no return values
computeHrsCompleted()
—no return values
computeCreditsEarned()
—return values
computeCumulativeGPA()
—no return values
computeCumulativeHrsAttempted()
—no return values
computeCumulativeHrsCompleted()
—no return values
computeCumulativeCreditsEarned()
—no return values
PrepareReport()
—no return values

FIGURE 8. Composition of Student Grade Report.

An example of a function activated as a result of an event can be found in the event Student Registers for Class (see Figure 3 for the event script). The functional request Student Requests for Student Class Schedule activates the function that produces the Student Class Schedule. The function is activated when the event Student Registers for Class happens. Figure 9 shows the function script for the functional request Student Requests for Student Class Schedule.

5. CONSOLIDATION OF OBJECTS IN THE SYSTEM

By identifying events and functions, the previous three steps have assisted us in identifying domain objects, supporting objects and agent objects for an application domain. Relationships among these objects have also been identified, albeit in a very segmented manner. In step 4, we consolidate the information gathered from the previous three steps, refine the definition of objects and identify further services required to support and facilitate the activities of the application system.

The refinement of object definitions would involve the use of abstraction mechanisms such as *generalization–specialization* and the *aggregation* concepts (see Jacobson, 1992). In addition, a dictionary of object definitions will be compiled in this step. The structure of an object definition includes the following information:

- Name of object.
- Type of object.
- Names of objects from which attributes and behaviour are inherited.
- Names of objects which inherits its attributes and behaviour.
- Attributes identified with the object.
- In the case of domain objects, actions of the object.
- Class services and instance services provided by the object.
- In the case of domain objects, the domain policies relating to its actions.
- Relationship attributes with other objects in the application domain.

Figure 10 shows the consolidated definition of the Student class from the previous steps.

5.1. State transition diagrams

For each domain object identified, a state-transition diagram is also constructed. This diagram shows the local dynamic behaviour of domain objects. A state-transition relates the present state of a domain object to its possible future states. By considering the state-transitions, we would more accurately define a domain

Function S.1f Student Requests for Student Class Schedule

Data: student number
From: event S.2e or S.3e
Temporal Info: STUDENT has finalised his registration for the term

Meaning:
The Student Class Schedule of a student can be printed out either through the events S.2e or S.3e

Activity:
1. Verify the existence of SCHEDULE using student number
2. Compute the total_hours for the courses registered for the term using the credit_hours for all courses in SCHEDULE
3. Format the SCHEDULE
4. Output the formatted SCHEDULE

FIGURE 9. Function script for Student Requests for Student Class schedule.

object since a domain object does not conduct its activities in a random manner in the real world.

Take for instance a Student object and a Class object in the Student-Records System. A student must declare

Object Class: Student **Type:** Domain object
Inherits From:
Inherits By: **Action:**
Attributes: Matriculate()
student number Register(class number, term id, section
student name number)
address CancelRegistration(class number, term id,
phone number section number)
major code Declare(major code, minor code)
[minor code] Grade(course grade)
ChangeGrade(course grade)
Link Attributes: Confer(date)
{ dept. id
class number **Class Service:**
term id StudentExists(student number)
section number —return Boolean value
course grade AssignNo()
}* —no return values
advisor code CheckGraduatingStudent()
—return a list of student number that are due to be conferred degree

Service:
MeetGrade(major, hours, grade)
—return Boolean value
CheckRegistration()
—return Boolean value
CheckCourseGrade(class number)
—return class number and grade
DisplaySchedule()
—no return values
GetStudentInfo()
—return details of student
GetRegisteredCourses(term id)
—return list of courses registered for the term
TakeClass(class number, term id, section number)
—return Boolean value
UpdatedGrade(class number, term id, section number)
—return Boolean value
GetGrade(class number)
—return grade
PrepareDeclaration()
—no return values

FIGURE 10. Definition of domain object, Student.

his major (and minor) before he can register for classes; a class cannot enrol/register students unless it has been offered for the term. Figure 11 shows the state-transition diagram of the domain object Student in the Student-Records System. It shows the set of states that Student can take and the actions that would bring it to a particular state given a present state. Any action that do not conform to any of the state-transitions is considered as invalid. A valid state-transition for Student is (REGISTERED, Grade, GRADED), i.e. if a student object is in the state REGISTERED and the action is grade, then the object will proceed to the GRADED state upon the successful completion of the action.

A state-transition diagram records the entire life history of an object. Since an object's life history consists of a beginning and an end, we may need to add further actions to the set of actions derived from event scripting in the earlier step. Consider as an example the Student object. From event scripting, we have identified register, cancel, declare and grade as the actions of Student. However, we note that a student's life history in a university begins when he is matriculated into the university and ends when he graduates from the university. Therefore, two additional actions, matriculate and graduate are necessary and they are added into the life history of Student.

A consolidated object-relationship diagram is produced from the set of segmented object-relationship diagrams specified in the earlier steps. Figure 12 is a first-cut consolidated object-relationship diagram for the Student-Records System. The object-relationship diagram will be refined using abstraction mechanisms like generalization–specialization and aggregation. The refinement process will be further elaborated below.

5.2. Aggregate objects

An aggregation is a kind of association between a whole and its parts in which the whole is composed of the parts. The aggregation relationship is a special form of relationship with the following properties (Rumbaugh *et al.*, 1991):

- Transitivity—if A is part of B and B is part of C, then A is part of C.

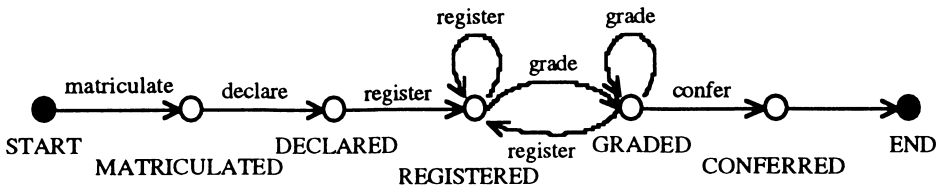

FIGURE 11. State-transition diagram for domain object Student.

- Anti-symmetry—if A is part of B, then B is not part of A.
- Some properties of the assembly propagate to the components as well, possibly with some modifications.

The relationships between Class, Pre-requisites and Section are refined by making Class as an aggregate of Pre-requisites and Section. Pre-requisites is made a component of Class because it has a tight relationship with respect to Class (see Figure 13). The latter has the responsibility to ensure that a Student object registering into the class meets its prerequisites. Thus, the characteristics of Class that is connected to its prerequisites is delegated to its component, Pre-requisites. Making Pre-requisites a component of Class would thus better reflect the class association between the two objects. As can be seen from Figure 13, Pre-requisites is a supporting object.

5.3. Generalization–specialization

Identified classes are organized into a hierarchical classification tree with classes higher up in the tree being superclasses of those classes lower down the tree. The latter are known as subclasses. A superclass is a generalized class of its subclasses (and the subclasses are specialization of the superclasses). However, objects are organized into a classification tree only when they are semantically related. Assume that the student population in the university of which the Student-Records System is a part is differentiated into under-graduate and post-graduate students. Both categories of students are semantically related in that they are students of the university. They share certain common properties. However, there are other properties that are unique to a particular category; for example, the registration criteria for under-graduates are different from those of the post-graduates. Therefore, the Student in the Student-Records System could be defined as a superclass of Under-Graduate Student and Post-Graduate Student as shown in Figure 14.

6. DYNAMIC BEHAVIOUR SPECIFICATION THROUGH USE SCENARIOS

In this step, the dynamic behaviour of a system is specified. This is carried out by considering how a system will be used by its users. The purpose of this step is to understand the interactions among the objects, to identify further objects, where necessary, and enhance on the set of operations defined for the objects. If the modelling process in the earlier steps had identified all the domain objects and actions, then this step would lead to the identification and definition of services.

6.1. Use scenarios and interaction diagram

To understand object interactions, we shall use interaction diagrams and use scenarios to help us describe how a user would use the system. A use scenario describes a behaviorally related sequence of activities in a dialogue that a user has with a system. A use scenario is similar to what Jacobson called a use case (Jacobson, 1992). A user is one who uses a system and is different from an external entity in a context diagram. For example, a student is an external entity of the Student-Records System but he may not be the user. An external

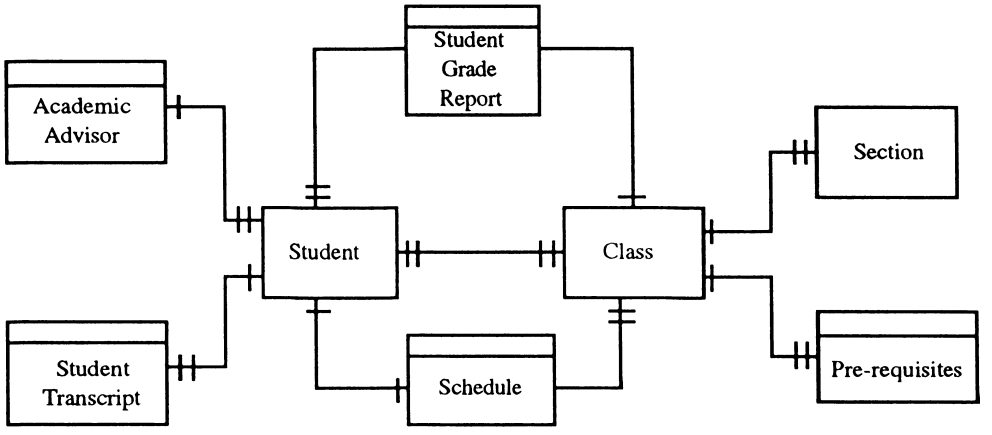

FIGURE 12. First-cut object-relationship diagram for a Student-Records System.

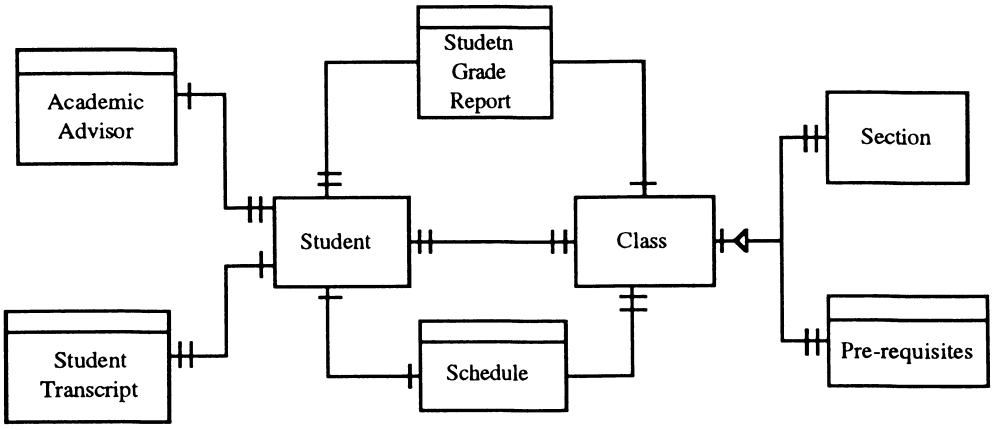

FIGURE 13. Consolidated object relationship diagram for a Student-Records System.

entity is one who triggers events and make functional requests on a system whereas a user is one who uses a system; the latter is a facilitator of events or functional requests. An example of a user is the data-entry clerk in the registrar's office.

A use scenario is different from an event or function. It may encompass one or more events or functions. For example, the use scenario *student registration* consists of an event Student Registers for Class and the activation of a function that serves the request Student gets Student Class Schedule.

A list of *use scenarios* for a system is first drawn up. Each use scenario is described using a use scenario script. Figure 15 is a use scenario script for *student registration*. The information contained in the script includes:

- Name or identifier of the use scenario.
- The possible users of the use scenario, i.e. the interviewee(s) or references, if appropriate.
- The activation.
- The events involved.
- The functions involved.
- Objects involved.
- The description of the transaction sequence involved in the use scenario.
- Results/effects of the use scenario.

6.2. Port and interface objects

The interaction of objects involved in a use scenario is illustrated using an *interaction diagram* which is a graphical representation of the communications among the objects of a system. In this way, designers can, in principle, validate the structure of the use scenario (Jacobson, 1992). In Figure 16, we see that new objects such as Registration Window had been introduced in the interaction diagram. Registration Window is an example of a *port*. A port connects a user to a system. It is an interface between the real world and the system. It receives inputs from the users, calls on the respective objects with the inputs and channels whatever outputs the system may have for its user to the user. Ports are introduced in order not to let changes in the interface of a system affect the definition and design of domain, supporting or agent objects. This is necessary as changes to a system interface are common and should typically affect only the ports (Jacobson, 1992).

A port may be decomposed into its constituent objects known as *interface objects*. The levelling concept of the data flow diagramming technique (Yourdon and Constantine, 1979) is used to depict the composition of a port. In the levelled diagram, interface objects and their interactions will be depicted. For example, the port Registration Window is decomposed into a set of

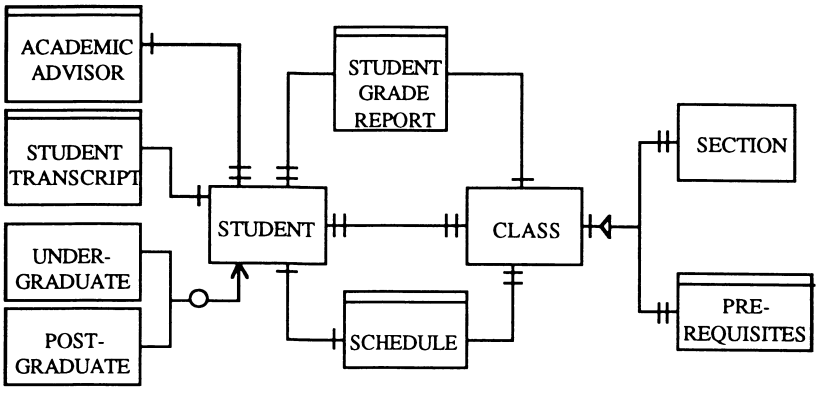

FIGURE 14. Object relationship diagram for a Student-Records System.

Use Scenario Student Registration

Users: Clerk
Activation: None
Event: S.2e—Student register for Class
Function: S.1f—Student Requests for Student Class Schedule
Domain Objects: STUDENT, CLASS
Supporting Objects: SCHEDULE, PREREQUISITES, SECTION
Port Object: Registration Window

Meaning:
STUDENT may register for a certain number of CLASSES, specifying the SECTION of the CLASS he wishes to register for, provided:
i) the STUDENT meets the PREREQUISITES for the CLASS (if any)
ii) there are seats available for that CLASS
—if the SECTION that he wish to register for is filled, i.e., seats available = 0, alternate SECTIONS of the CLASS may that do not have conflicts with the requested SCHEDULE be offered to him for selection (if there is any)
—if there is conflict with the requested SCHEDULE, alternate SECTIONS of the CLASS that do not have conflicts with the requested SCHEDULE are offered to him for selection
Maximum number of CLASSES that STUDENT must register for per term = 8
A student class Schedule will be generated when registration process is completed.

Activities:
1. Clerk give command to register STUDENT for CLASSES
2. The Registration Window is displayed
3. Clerk enters student number
4. Check if STUDENT with student number exists
5. Get student information using student number
6. Creates Schedule for STUDENT using student information
7. Display student information
8. Repeat{
Clerk enters class number and section number
Check if SECTION of CLASS with (class number, section number) exists
STUDENT register for (class number, section number)
STUDENT ensures number of CLASSES registered $\leqslant 8$
CLASS register STUDENT
CLASS ensures Pre-requisites are met by STUDENT
CLASS ensures seats availability
checks SECTION for seats availability
checks if SECTION's seat conflict with SCHEDULE
if conflict, suggest alternative
Given alternative, get confirmed choice
SECTION registers STUDENT using student number
STUDENT complete register action
STUDENT calls SCHEDULE to display the registered CLASS
SCHEDULE calls Registration Window to display the registered CLASS
} until STUDENT has registered for all the CLASS for the term
9. Clerk gives command to print out the Student Class Schedule
10. Schedule prepare format of Student Class Schedule
11. Printer print student class schedule

Result/Effect: Output Student Class Schedule

FIGURE 15. Use scenario script of Student Registration.

interface objects such as a Menu object, a Scroll Bar object, a Selection List object, etc.

Figure 16 is an interaction diagram for the use scenario Student Registration. Each participating object in a use scenario is represented as a column in the interaction diagram. The order of the columns is insignificant and the columns are ordered in a manner that enables a clear interpretation of the interaction diagram. If the interaction diagram involves several instances of the same class, we can either draw different columns representing the different instances or use the same column to represent them depending on the legibility of the representation. All the behaviour of an object in the use scenario is attached to the column representing the object.

The interaction diagram is similar to that used in the Jacobson's method (Jacobson, 1992) except that the different types of operations (actions, class services or instance services) are distinguished from one another. A block in an interaction diagram is representative of an operation. A shaded block is an action. In Figure 16, the shaded blocks for Student, Class and Section represent the Register action of the three objects.

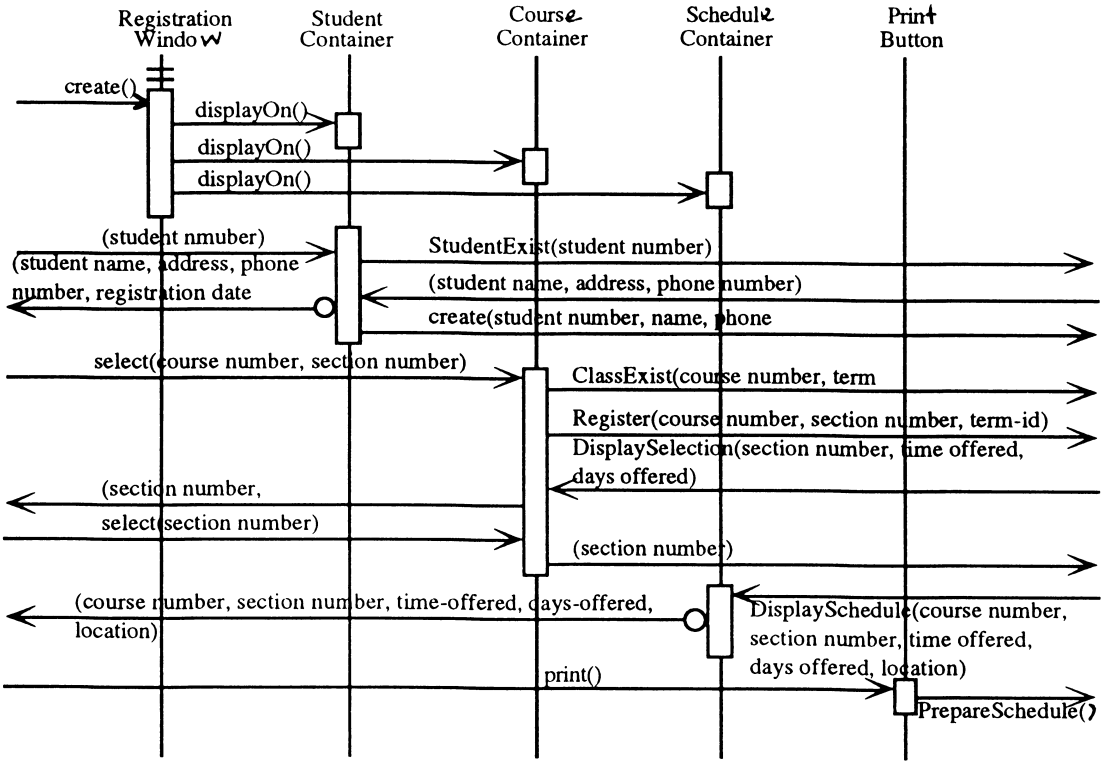

FIGURE 16. Levelled object interaction diagram for Registration Window.

6.3. Messages

Objects communicate with one another by sending messages. An arrow in an interaction diagram is a message sent from one object to another. A message is associated with an operation. There are three types of message.

The first type of message is indicated by a regular arrow. It is an activation of an operation which could be an action or a service.

The second type of message, represented by an arrow with an empty arrowhead, is an activation of a service prior to the execution of an action. The activated service is connected with a pre-action condition check. For example, prior to the registration of a student into a class in the Student-Records System the condition 'the student must not have already been registered for 8 classes' must be satisfied. We express this in the interaction diagram by a message sent to a service called CheckRegistration(). With this type of message, we can express the fact that the action Register can only take place when the condition [realised through the service CheckRegistration()] is satisfied.

The third type of message, represented by an arrow with a darkened arrowhead, is an activation of a service after the successful completion of an action. For example, in Figure 16, after the completion of the action Register of Student, a service DisplaySchedule() is triggered. The service will display the information related to the class a student is registered for. It is thus a post-action service.

6.4. Class and instance services

In an interaction diagram there are class services and instance services. A striped block is representative of a class service whereas an empty block is an instance service. A data flow is indicated in an interaction diagram by a broken arrow. There are times when we want to represent a data flow out of the system that does not require a user's response to it. We represent this type of data flows by a broken arrow with a circle at its tail.

6.5. Decomposing port

The decomposition of the port begins with an initial screen design. This is done through interactions with the users of a system. Figure 17 shows an example of a screen design. Using the screen design, a port such as Registration Window is decomposed into their constituent interface objects. Figure 18 is an object-relationship diagram showing the relationships between the port, Registration window, and its constituent interface objects. Figure 18 shows the result of the decomposition process for Registration Window.

7. CONCLUSION

On the outset, it is mentioned that a software development method is necessary to guide developers in a systematic step-by-step approach to constructing systems. Many object-oriented methods have been proposed over the years. They generally address two

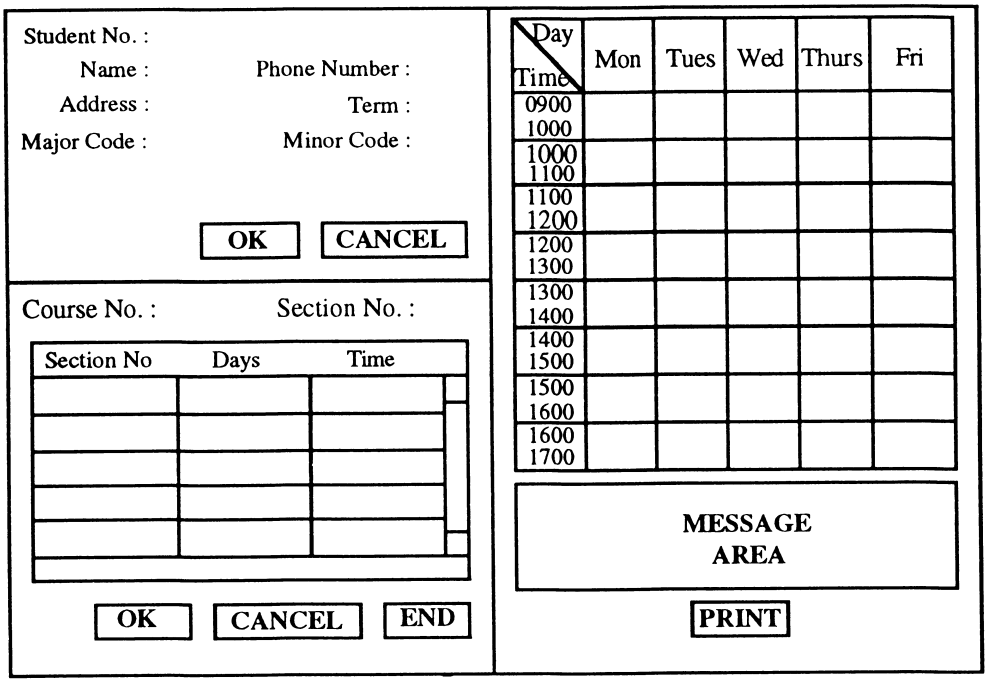

FIGURE 17. Screen design.

concerns in the application of the technique: object model construction and software maintainability. However, it was felt that functional requirement changes to a system are not well addressed in the methods.

In this paper, an object-oriented development method based on the approach advocated by the Jackson method is proposed and discussed. The modelling process of the proposed method separates the real world subject matter about which a system is to compute from the functions to be provided by a system for its users. In this way, changes to the volatile functional requirements on a system will not have adverse effects on the other parts of a system that have little or no connection with the changes. Through this approach to modelling and constructing systems, it is hoped that the overall maintainability of the system could be improved beyond what the object-oriented paradigm could offer. Evaluation of the proposed method is presently being carried out.

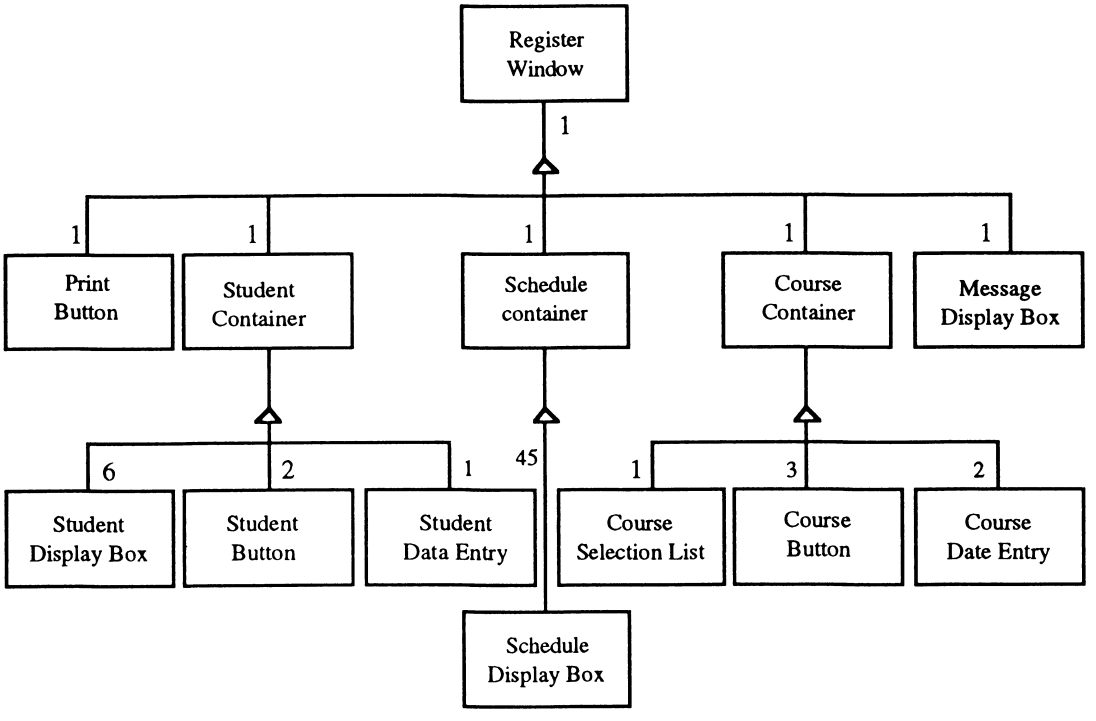

FIGURE 18. Registration Window and interface objects.

ACKNOWLEDGEMENTS

The authors would like to acknowledge the financial support granted by the National University of Singapore (grant RP910689). They are grateful to the University in granting them the opportunity to carry out the work as described in this paper.

REFERENCES

Booch, G. (1991) *Object-oriented Design*. Benjamin/Cummings.

Jackson, M. A. (1983) *Systems Development*. Prentice-Hall.

Jacobson, I. (1992) *Object-Oriented Software Engineering*. Addison-Wesley, Reading, MA.

Layzell, P. J. and Davis, C. G. (1993). *The Jackson Approach to System Development: An Introduction*. Chartwell Bratt.

Martin, J. (1983) *An Information Processing Manifesto*. Savant, Carnforth, UK.

Martin, J. and Odell, J. J. (1992). *Object-Oriented Analysis and Design*. Prentice-Hall, Englewood Cliffs, NJ.

Meyer, B. (1988) *Object-oriented Software Construction*. Prentice-Hall, Englewood Cliffs, NJ.

Rumbaugh, J. *et al.* (1991) *Object-Oriented Modelling and Design*. Prentice-Hall, Englewood Cliffs, NJ.

Weiss, S. (1989) *Object-Oriented Analysis and Design: The Synthesis Model*. Wayland Systems, Seattle, WA.

Wirfs-Brock, R. and Wilkerson, B. (1989). Object-oriented Design: a responsibility-driven approach. In *Proc. OOPSLA '89*, pp. 71–75.

Yourdon, E. and Constantine, L. (1979) *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Yourdon Press.