Joint Conference of 1996 International Computer Symposium
December 19~21, Kaohsiung, Taiwan, R.O.C.

OBRAD: An Object-Based Rapid Application Development Method*

Wei.T. Huang, Jonathan Lee and Tai Lin Kuei

Institute of Computer Science and Information Engineering
National Central University
Chung-Li, Taiwan 32054
E-mail: wthuang@csie.ncu.edu.tw

Abstract

For almost every application, user's needs are constantly evolving. Thus, the system being constructed is always at a moving target [6]. This is primary reason for that software fails to meet customer expectations. System development methods must take into account that users, and their needs and environment, change during the process. One of the best ways is that software is constructed by using an evolutionary paradigm, by which software maintenance may take place in the specification and/or design phases. We propose in this paper a software development process based on object-oriented concept in a rapid prototyping style. The development process consists of: (1) statement of requirements; (2) specification/design; (3) construction; and (4) validation. This method is an integrated approach by using JSD, VDM, Warnier/Orr diagram, automatic code generation technique, and visualizing software for validation.

Key words: *Software engineering, Software lifecycle, Object-orientation, Jackson System Development (JSD), Vienna Development Method (VDM), Warnier/Orr diagram, Rapid application developmemt (RAD), software visualization, CASE tool.*

1 Introduction

The concept of the traditional phased refinement approach of software development (life cycle model) perpetuates the failure to build an effective environment for the communication between end-user and system analyst [15]. This is due to three reasons [9]:

- There is usually a significant culture gap between the user and the developer and the way they communicate [5].
- The user who is unfamiliar with information technology may have produced very vague requirements which could be interpreted arbitrarily by the developer.
- The changes of requirement during development is difficult to be truly reflected in the system specification because of the communication gap between user and developer. As a result, a system meets the requirements is a risky and error prone activity.

Unfortunately, even when a software developer uses modern notations and techniques success is likely only when the application is both well understood and supported by previous experience [4]. A software system is constructed step-by-step through the phases of requirements, specification, design and implementation. People and developers can operate the system only when the whole system is completed. Maintenance is always done on the low level. The activity from requirements to implementation should be repeated in order to ensure the quality of modifications. Software developement process is therefore accepted to be the high cost of error correction, especially errors done in requirements and specification phases. This kind of software development approach is considered harmful [15].

In this paper, we propose an alternative development process that a software system is defined at first in small, and a rapid prototype is evolutionary developed so that the user can interact with it and validate or change the requirements if needed. By developing and delivering working prototypes of application, the expectation gap between user and developer is repeatedly closed, so that the distance between what is expected and what is delivered is negligible. This way of development approach not only can manage the changes of the requirements of poeple and environment of the system, but also can prevent the propagation of errors from specification phase.

Figure 1 shows a process model of the evolutionary paradigm [19] that can promise the progress of software development. As can be observed, the early specification and design phases are unified and the later coding and test phases are eliminated from the traditional software development process.

Requirements and specification, as the principal activity in this paradigm, is very important for developing reliable, high quality and usable systems. Developers can concentrate themselves on 'what' in-

*This work was partially supported by National Science Council, Republic of China, under Grant NSC85-2213-E-008-005

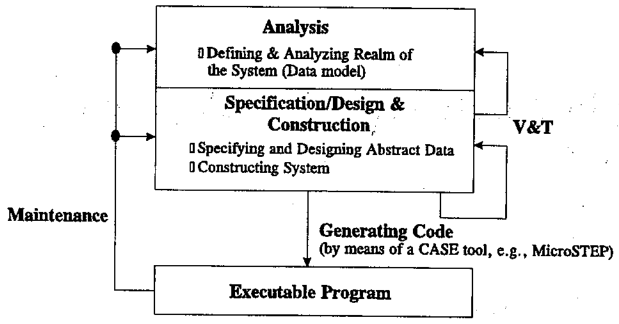

Analysis
Defining & Analyzing Realm of the System (Data model)
Specification/Design & Construction
Specifying and Designing Abstract Data
Constructing System
V&T
Maintenance
Generating Code (by means of a CASE tool, e.g., MicroSTEP)
Executable Program

Figure 1: An Evolutionary Paradigm for Software Development.

stead of 'how' to do in the software development activity. Maintenance is done on the highest level by editing the specification. Moreover, the executable applications can then be built with incomplete specification (i.e., an evolutionary prototyping technique). Another important issue is that, in the evolutionary paradigm the reuse of system units is more abstract, that is, the reuse of specification or design. This can dramatically increase productivity and quality.

In this paper, we use the modeling technique of JSD [11] to define and to analyze the (kernel) realm of a system, and use abstract data type as the backbone of object-oriented software development. Software functionalities are expressed by an extended Warnier/Orr diagrams, and system structures are described by SoftVision graphical notations. The specifications of the entities which are identified by using JSD modeling technique are written by VDM notations. The Warnier/Orr diagram can be converted to pseudo codes and then to any programming language. It can also be directly transfered to the graphic language, e.g., MicroSTEP [20], by which generating applications is incredibly simple.

2 An Object-Based Software Development Method

We will describe in this section a software development method which is on the basis of the data inherent in an application. The whole development process will be driven by the idea of object orientation:

1. *Statement of requirements.* The purpose of the system is briefly stated.

2. *Entity analysis.* The realm of the system is defined by using JSD modeling technique. Entity and its basic operations are specified.

3. *Specifying/designing the abstract data.* Entity and the basic operations are formally specified and designed by the Vienna Development Method.

4. *Constructing the system.* The software system is constructed by reusing the basic operations specified in step 3 by means of the extended Warnier/Orr diagram.

5. *Code generation.* Pseudo codes are written and converted to a programming language, or an executable program is automatically generated on the basis of the system structure constructed in step 3 by means of a CASE tool.

6. *Visualizing software.* Software system can be validated by means of SoftVision notations.

In order to explain this development process more in detail, we borrow an example of a flight booking system from [10].

2.1 The Statement of Requirements

A Flight Booking System: An airline company schedules a number of flights between its Taipei base and the main cities in Taiwan in one type of commuter aircraft. The system will be operated by booking staff. Customers may take a booking through telephone or at an airport. The system will allow a booking staff to do the following task:

- To book a flight for a particular passenger who should provide his/her name, address, and flight date.

- To provide a listing of passengers who are currently booked on a flight.

- To cancel a particular flight that has already been booked for a particular customer.

- To check whether there is an available seat for a particular customer.

The system must satisfy some constraints. For simplicity, these constraints will not be explicitly reflected in our system design.

2.2 The Entity Analysis

The first stage in the development method we propose in this paper is the identification of entities based on the modeling technique of JSD. An entity is an object of interesting in an application which has associated with it a set of data and time ordered actions. An action is an atomic event that operates on an entity during a moment in time which cannot be decomposed into any further actions. An action must be relevant and detectable. It will cause a response in the system and eventually cause a database update or other change in system state. There are two other concepts which are also important: that of *entity attributes* and *action attributes.* The former are descriptions of an entity; the latter are input data messages consumed by actions. Entity attributes are created and updated by an entity's actions and are internal variables that can be thought of as a kind of status record of an entity, that is, entity attributes (or state vectors) describe the state of the entity. Since actions respond to events in the real world, these events are communicated to the system as data messages and are refered to as the attibutes of an action.

Identification of Entities and Actions

1. Identifying entities. To identify the entities in a system, there are three questions we should ask [16]: what application problem is the system to solve, what is the goal the system aims to achieve, and what are the objects the system responds to or manipulates? For example, in the Flgiht Booking System, the application problem that to be solved is to have a system that administers the bookings of flights in Taipei base. There are a number of goals: to allow a booking staff to book a flight, to list passengers who are currently booked on a flight, and to cancel a particular flight. In the system, *booking* is the entity that manipulates and *customer* is the entity that responds to. Both entities carry out a series of actions that are associated with action attributes, and these actions either read or update the attributes of the entities.

2. Identifying actions. There is a four-step process that should be adopted to identify the actions in a system [16]:

    - Identify all the external events which happen in the real world.
    - Find out how these events are communicated to the system as input.
    - Identify the system inputs, these will be action attributes.
    - Identify actions which are necessary to respond to the messages contained within the action attributes.

    In order to describe this four-step process more in details, consider again the Flight Booking System stated above. The external events that could occur in such a system is as follows: the system allows a booking staff to book a flight for a particular passenger over phone or at an airport, the particular flight that has already been booked to a particular customer could be canceled by the booking staff, and the flight date and destination could be amended on customer's request, the system allows the booking staff to provide a list with flight number and date of passengers who are currently booked on a flight. In this example, there are obviously four entities: a FLIGHT, a SEAT, a CUSTOMER, and a BOOKING. Tying the actions and their attributes associted with the entities together, we document the following relationships partially:

    <u>**Entities**</u>

    - FLIGHT. Attributes: flight number, destination, date, time of departure.
    - CUSTOMER. Attributes: customer name, customer address, customer phone number.
    - BOOKING. Attributes: flight number, date, customer name, customer address, customer phone number.
    - SEAT. Attributes: seat number.

    <u>**Actions**</u>

    Entity CUSTOMER

    - *Create.* Customer books a flight. Attributes: customer name, customer address (with phone number).
    - *Amend.* Customer changes his/her address. Attributes: customer name, customer old address, customer new address.
    - *Purge.* Customer cancels a flight. Attributes: customer name, customer address.

    Entity BOOKING .

    - *Create.* Booking staff books a flight. Attributes: flight number, date (including departure time), destination, customer name, customer address.
    - *Cancel.* Booking staff cancels a particular flight. Attributes: flight number, date, customer name, customer address.
    - *Archive.* Booking staff archives the booking for customer. Attributes: flight number, date, customer name, customer address.

3. Ordering actions in time. The next stage is to describe the entity life history (ELH) by ordering actions in time. The Jackson's graphical notation is used for this purpose. For our example, the ELH of the flight and seat entities are not required since they do not carry out the creation, deletion and amendment of the entity attributes. The ELH of a CUSTOMER and that of a BOOKING are shown in Figures 2 and 3.

4. Specifying basic operations. The final and important step in entity analysis is to create a bridge between entity analysis task and the process of data design. In order to do this the low-level basic operations should be identified. Basic operations which are decorated with smaller, numbered boxes will be in some case in one-to-one correspondence to the actions with which they are associated. When an action is carried out a number of basic operations are executed. The basic operations of CUSTOMER and BOOKING are also shown in Figures 2 and 3.

The basic operations will eventually be implemented as subroutine calls on stored objects and provide input into the design stage. In our Flight Booking System the basic operations for the entity CUSTOMER, and the entity BOOKING are listed as follows:

<u>**List of the Basic Operations**</u>

Entity CUSTOMER

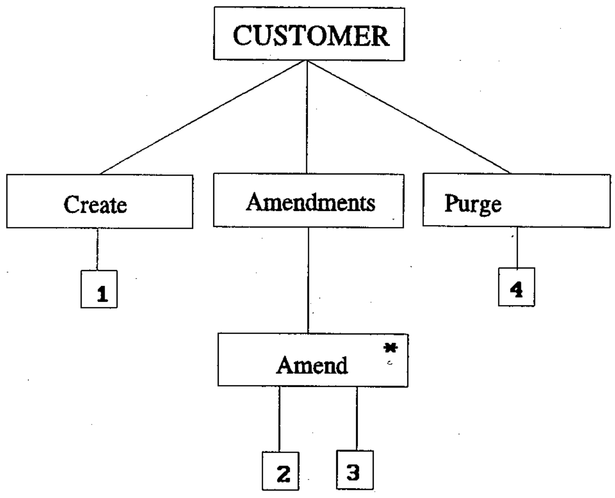


Figure 2: The Life History of a CUSTOMER with Basic Operations.

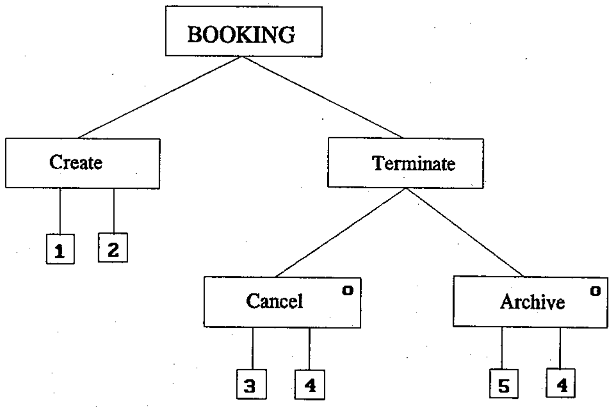


Figure 3: The Life History of BOOKING with Basic Operations.

1. *createCust (custname, custaddr)* creates a customer with a name *custname* and address *custaddr.*

2. *checkCust (custname, custaddr)* checks whether a customer with a name *custname* and *custaddr* is valid.

3. *modifyCust (custname, oldaddr, newaddr)* amends a customer's old address *oldaddr* to new address *newaddr* with a name *custname.*

4. *deleteCust (custname, custaddr)* expunges the customer with a name *custname* who has an address *custaddr* from the system.

Entity BOOKING

1. *checkSeat (fltno, fltdate, destination)* checks whether there is a seat available with flight number *fltno* on flight date *fltdate* to *destination.*

2. *bookSeat (fltno, fltdate, destination, custname, custaddr)* books a seat with *fltno* on *fltdate* to *destination* for the customer with a name *custname* and address *custaddr.*

3. *checkBooking (fltno, fltdate, custname)* checks that there is actually a booking made for the customer with a name *custname* on the flight with *fltno* on *fltdate.*

4. *deleteBooking (fltno, fltdate, custname, custaddr)* deletes the booking for the customer with a name *custname* and *custaddr* on the flight on *date* with *fltno.*

5. *archiveBooking (fltno, fltdate, custname, custaddr)* archives the booking for customer with a name *custname* and *custaddr* on the flight with *fltno* on *fltdate.*

2.3 Specifying Abstract Data

The next stage is to design abstract data type that corresponds to the entities. An abstract data type (ADT)[1] is a collection of data and the operations on that data. The specification of an ADT concentrates on describing data by the collection of operations that operate on it.

The Vienna Development Method (VDM) originally used the Meta-IV notation as described by Jones [12] and Bjorner & Jones [3] to describe a software system in terms of abstract data types. In this way, software is specified in purely logical terms without considering its actual machine representation. In VDM a software system is characterized as a set of modules which each have an internal state consisting of a co-existing set of variables. Variables may be typed using primitive or ADT, which can be regarded as the equivalence of a class in the concept of object-orientation. The operator symbols are illustrated as follows:

Operations on sets	
$\in$	is a member of
$\cap$	intersection
$\cup$	union
Operations on mappings	
$\xrightarrow{m}$	maps to
$\cup$	union
$\dagger$	overwrite
$\setminus$	restriction by
dom	domain of
Logic noations	
	~not
$\Leftrightarrow$	is equivalent to
$\wedge$	and
$\vee$	or

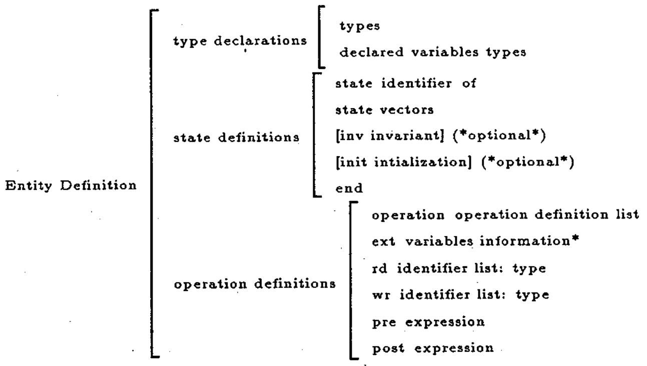
[1]In this paper an ADT is the synonym of an object. Other developement methods call it an abstract machine [1].

Some keywords:

ext	external
rd	read only
wr	read/write
mk- funtion	composite ADT instantiation
inv	invariant
init	intialization

An entity can be seen as an ADT or an object class in our approach. We use the VDM notations to specify the entity and the basic operations as the following specification document [2].

Entity Definition



For example, the specification of the entity CUSTOMER is shown below:

CUSTOMER

```
types
maxbookings =            50
Seatallocations =        Seatnumber ─m→ Person
Person::                 name: Name
                         addr: Address
Seatnumber =             N
Name =                   String
Address =                String
state CUSTOMER of
         customer: Person
inv      dom Seatallocations ≤ maxbookings
end

operation createCust (custname: Name, custaddr: Address)
ext     wr customer: Person
pre     mk- Person(custname, custaddr) ∉ dom customer
post    customer = customer↼ ∪ mk-Person(custname, custaddr)


operation checkCust (custname: Name, custaddr: Address) r:  B
ext     rd cust: Person
post    r ⇔ mk-cust (custname, custaddr) ∈ customer
```

```
operation ModifyCust (custname: Name, oldaddr: Address,
        newaddr: Address)
ext     wr customer: Person
pre     mk- Person(custname, oldaddr) ∈ dom customer
post    customer = customer↼ ∪
            mk- Person(custnm, oldaddr) † mk-Person(custname,
            newaddr)

operation deleteCust (custname: Name, custaddr: Address)
ext     wr customer: Person
pre     mk- Person(custname, custaddr) ∈ customer
post    customer = customer↼ - mk- Person(custname, custaddr)
```

The entity specified in this way can also be seen as an *abstract machine.* The abstract machine is used to draw attention to the fact that a specification of a data type, together with the operations on that data type, is rather like a specification of a machine. It is something like a real machine in that it has bottoms and knobs which can be used to trigger internal state changes, and it has a way of displaying information about the internal state. For both real and abstract machine, we should not be concerned with the exact details of how they work, but only with what functionality is provided by them. An abstract machine can well represent any real-world object that is being modelled in the system. This is what object-oriented design is about. Abstract machine can be operated by a user via some sort of interface. In this paper, we apply this concept to construct our software.

Given this kinds of specifications the various entities can be implemented by the object-based or object-oriented programming language such as Ada, Modula-2, or C++, or eventually a business language system: MicroSTEP [19].[3]

2.4 Specifying the Functionality

The functional specification to be available at a later stage of software development has an effect that few changes will impact on it. It is the principle that object-oriented development allows the developer to delay considerations of detailed functionality until late in software development, though an outline of functional specification may exist for costing estimation, and rough notes for anlysis purposes. The notation we use to describe the specification is Warnier/Orr diagram [14] because it is a popular and compact graphical notation which describes the system construction. In this paper, we use an extended Warnier/Orr diagram to construct the system by the basic operations specified and designed in step 1 and step 2.

The Warnier/Orr diagram enables the designer to represent the software structure hierarchically in a compact manner, and can be easily translated into pseudo program. The main tool in a Warnier/Orr diagram is the brace '{', also called universal which

[2]VDM can be applied not just to the specifications, but also to the design and implementation. For the syntax of VDM laguage specification, readers may refer to [13] or [1].

[3]MicroSTEP is a trademark of SYSCORP International, Inc. It is an executable specification language system, by which software specification is specified by graphs.

shows decomposition of the system it depicts. Items that do not decompose further are called elements. If a data structure is represented by a Warnier/Orr diagram, the elements are data elements; and if a process of a system is expressed, then its elements are basic operations. Various data and process structure can be expressed by Warnier/Orr diagrams. The following is the extended Warnier/Orr notations used in this paper:

1. Sequence.

$$\text{program}\left\{\begin{array}{l}\text{Begin}\\ \text{[entity1.]operation1}\\ \text{[entity2.]operation2}\\ \text{[entity3.]operation3}\\ \text{End}\end{array}\right.$$

2. Selection.

$$\text{program}\left\{\begin{array}{l}\text{Begin}\\ \text{operation1}\\ \oplus\\ \text{operation2}\\ \text{End}\end{array}\right.$$

3. Repetition.

$$\text{program}\left\{\begin{array}{l}\text{Begin}\\ \text{operation(n, m)}\\ \text{End}\end{array}\right.$$

4. CASE structure.

$$\text{program}\left\{\begin{array}{l}\text{Begin}\left\{\ \text{get case value}\right.\\ \text{case value}\left\{\begin{array}{l}\text{case\_val1 : operation1}\\ \oplus\\ \text{case\_val2 : operation2}\\ \oplus\\ \text{case\_val3 : otherwise}\end{array}\right.\\ \text{End}\end{array}\right.$$

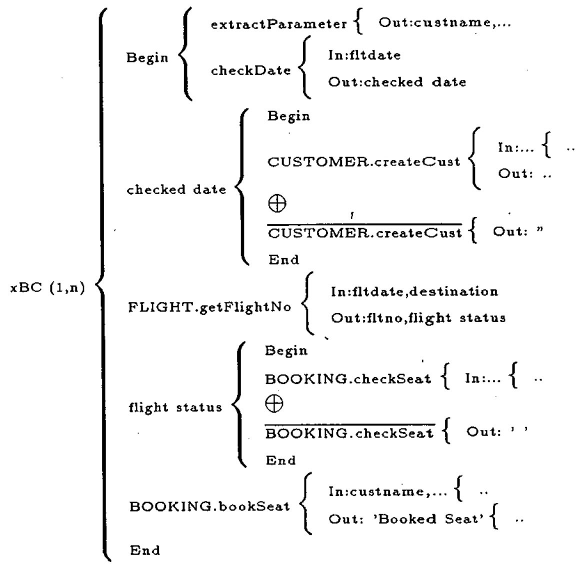
5. Input/Output.

   Syntax: operation (in: argument-1,...,argument-n; out: argument; inout: argument), where 'in' represents that the argument is the input, 'out' the argument the output, and 'inout' represents that the argument palys the role of input as well as output. Arguments are separated by comma.

6. (* *comment* *)

Here is an example to show the structure of the execBookingCommand modules by applying the extended Warnier/Orr notations. The other examples including Flight Booking System, execCancellingCommand, execAmendingCommand, execListingCommand, and execFinishingCommand are constructed in the same way:
Example: execBookingCommand (xBC)

- The Warnier/Orr Diagram



3 Code Generation Using CASE Tool

An alternative tool used to implement the software system may be the CASE tool called MicroSTEP. MicroSTEP promises the evolutionary software development paradigm. Systems specified by the tool is well modulized and highly modifiable. The principal operation of MicsroSTEP is to create system specification which can be directly transformed to executable programs. The operation consists of the following activities:

- Create the form of flow diagram which is constructed by data objects, process and links.
- Define the fields in each data object.
- Define the formats of reports and screens.
- Define the way to process input data objects to produce output data objects.

These activities are supported by a graphic based specification editors, namely, data structure builder, format builder, activity builder, and flow control builder. Moreover, the tool provides for the user the facility of reuse. When one wants to invoke another specification or database from current diagram, he/she can create a *call* to the existing specification or a copy to the existing database. Once the specification is created, MicroSTEP builds the working application through the following steps:

- Check specification for completeness and consistency.
- Generate code in C++.
- Compile and link the code.
- Install the application for specific working environment.

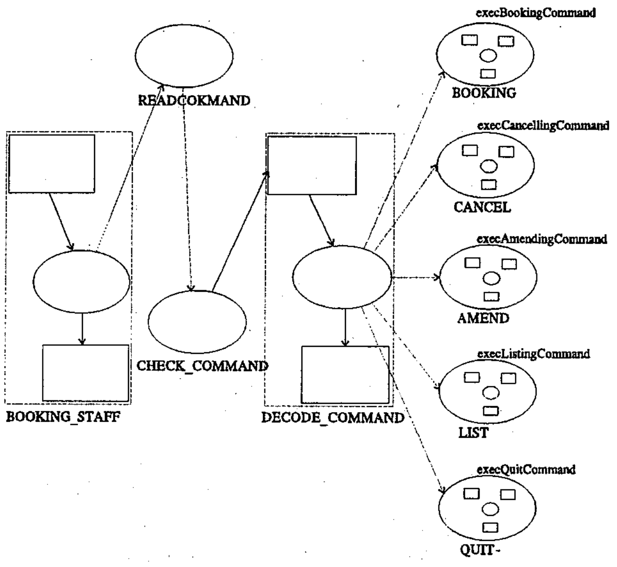


Figure 4: The Flow Diagram for Flight Booking System using MicroSTEP.

Readers may refer to [20] [18] for the detailed description of the tool. The Warnier/Orr diagram we specified in section 2 can be used as the 'concrete'specifications of the system for MircroSTEP. Figure 4 shows the flow diagram for the top-level of the Flight Booking System.

4 Visualizing Software

Here we introduce a graphical notations [2] called *SoftVision* which can be used to visualize software structure. It is a fully integrated approach that shows all the software development aspects at once. The user and developer are then easily "see software with mind's eye". SoftVision contains all of the parts of process, data, data flow, control flow, decision, parallel operation, and their interaction in one kind of diagram. The semantics of SoftVision is to put dataflow diagaram, state transition diagram, structure chart, flow chart and their interaction all together to help developer and user to analyze, design, and discuss software system.

The notation has the following advantages:

- It is simple, understandable, programmable and easy-to-learn.
- It facilitates mental simulation of software,
- It is able to aggregate pieces of software together on one diagram,
- It is an effetive communication tool for user and developer.
- It is able to express parallel process, and
- It can be tailored to specific kinds of software develpment method.

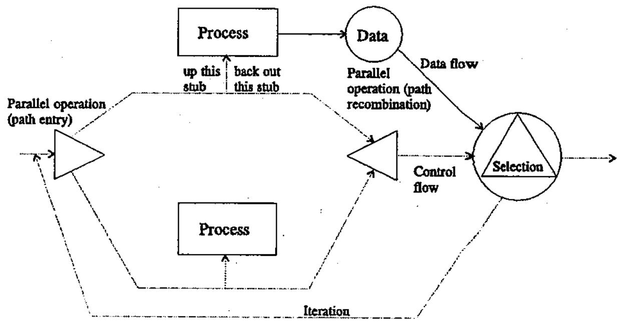


Figure 5: SoftVision Notations.

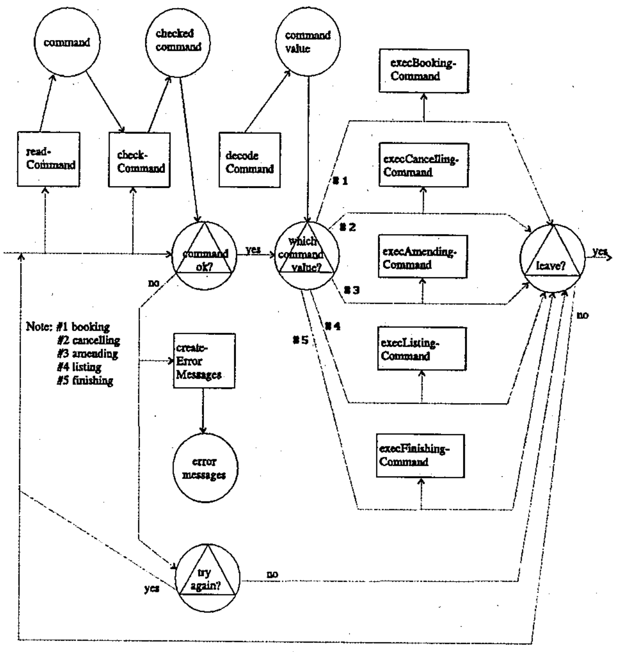


Figure 6: SoftVision Graph for Flight Booking System.

Some of these advantages, of course, are shared by other notations. Figure 5 shows the SoftVision notations in an "all-in-one" manner. Figures 6 visualizes the structure of the Flight Booking System. This graph can be used to facilitate mental simulation of the software.

5 Conclusion

An object-oriented system requires three major components [8] - abstraction, encapsulation, and polymorphism/inheritance. An object-oriented analysis and design must demonstrate these three components. We have demonstrated in this paper the concepts of abstraction and encapsulation except that of polymorphism/inheritance. The mechanism of polymorphism/inheritance will be required in the near furture.

The software development progression as shown in Figure 1 is a kind of "middle-out" approach. It is something like "making snowball". The phases to develop the system are: (1) requirements definition, (2) specification/design, (3) construction, and (4) validation. The advantages of this development paradigm are:

- The techniques we used are traditional, including JSD (modeling phase), Warnier/Orr Diagram, pseudo code, and CASE tool for code generation. They are integrated together to construct the system. These techniques are well established and well known by any software developer.
- VDM notations are used to formalize the specifications of the entities and their associated basic operations so that the specfications are defined unambiguously.
- The system can be automatically implemented by a CASE tool. In this way the development time can be dramatically shortened and a prototype of the system can be rapidly created for review.
- SoftVision is a kind of graphical tool that can be used to facilitate the mental simulation of software. We can use this tool to validate whether we have developed the right software. Though SoftVision is not a method, it can be tailored to specific kinds of software development method, such as we developed in this paper.

However, a complex large scale system is still to be developed in order to validate the evolutionary paradigm for the software development method. An inheritance mechanism is still needed to be established.

References

[1] Andrews, D. and D. Ince, *Practical Formal Methods with VDM*, McGraw-Hill, London, 1991.

[2] Bennet, W.S., *Visualizing Software - A Graphical Notation for Analysis, Design, and Discussion*, Marcel Dekker, Inc., New York, 1992.

[3] Bjørner, D. and C.B. Jones, *Foramal Specification and Software Development*, Prentice Hall, Englewood Cliffs, N.J., 1982.

[4] Blum, B.I. and R.C. Houghton, "Rapid Prototyping of Information Management Systems," ACM SIGSOFT Software Engineering Notes, Vol.7, No.5, 1982, pp.35-38.

[5] Christensen, K. and K. Kreplin, "Prototyping of User Interface," in *Approach to Prototyping* (edited by R. Budde, K. Kuhlenkamp, L. Mathiasen and H. Zullinghoven), Spring-Verlag, Berlin, 1984, pp.58-67.

[6] Davis, A.H., E.H.Bersoft and E.R.Comer, "A Strategy for Comparing Alternative Software Development Life Cycle Models," IEEE Transactions on Software Engineering, Vol.14, No.10, Oct. 1988, pp.1453-1461.

[7] Fertuck, Len, *Systems Analysis and Design*, Wm. C. Brown Publishers, 1992.

[8] Henderson-Sellers, B. *A Book of Object-Oriented Knowledge*, Prentice-Hall, Englewood Cliffs, NJ, 1991.

[9] Huang, W.T., "Rapid Software Prototyping Using Z and CASE Tool," in Proc. of NCS'93, Chiayi, Taiwan, 1993.

[10] Ince, D., *Object-Oriented Software Development with C++*, McGraw-Hill, London, 1991.

[11] Jackson, M.A., *Systems Development*, Prentice Hall, Englewood Cliffs, New Jersey, 1983.

[12] Jones, C.B., *Software Development: A Rigorous Approach*, Prentice Hall International, Englewood Cliffs, N.J., 1980.

[13] Jones, C.B., *Systematic Software Development using VDM*, 2nd Edition, Prentice Hall, New Jersey, 1990.

[14] Martin, James and Carma McClure, *Structured Techniques for Computing*, Prentice Hall, Englewood Cliffs, NJ, 1985.

[15] McCracken, D.D. and M.A. Jackson, "Life Cycle Concept Considered Harmful," ACM SIGSOFT Software Engineering Notes, Vol.7, No.2, 1982, pp.29-32.

[16] Sutcliffe, Alistair, *Jackson System Development*, Prentice Hall, New York, 1988.

[17] Warnier, J.D., *Logical Construction of Programs*, Van Nostrand Reinhold, 1974

[18] SYSCORP International, *MicroSTEP Reference Manual*, Version 1.6, SYSCORPT International, Inc., Austin, TX, 1992.

[19] Yeh, R.T., "An Alternative Paradigm for Software Evolution," in *Modern Software Engineering: Foundation and Current Perspective* (edited by P.A. Ng and R.T. Yeh), Van Nostrand Reinhold, New York, 1990, Chapter 1.

[20] Yeh, R.T., "MicroSTEP: A Business Definition Language System," in *Modern Software Engineering: Foundation and Current Perspective* (edited by P.A. Ng and R.T. Yeh), Van Nostrand Reinhold, New York, 1990, Chapter 17.