Describing Requirements in Lyee and in Conventional Methods: Towards a Comparison

Gregor v. BOCHMANN
School of Information Technology and Engineering, University of Ottawa, Canada

Abstract: The Lyee software development methodology has been recently introduced and suggests that the software development process can be largely simplified by starting with the capture of the user requirements and then generating the implementation code largely automatically, skipping in a sense the software design stage. This paper makes an attempt at comparing the notational concepts that underlie the Lyee methodology with the concepts that appear to be important for the description of software system requirements as defined by the main-stream software engineering methods. While we identify several relations between concepts in Lyee and corresponding main-stream concepts, we also point out a number of important differences, and we leave the reader with a number of questions which, we think, merit further discussion.

1. Introduction

It is well known that the development of larger software systems is a difficult and costly process. Many software development methodologies have been proposed to improve the efficiency of the process and the quality of the generated implementation code. It seems that one should not look for a "silver bullet" that provides a magic solution, but rather one has to look at many aspects of software development, including the choice of notations for describing system requirements, software architectures and designs, and the choice of the development process and the tools used during that process.

In this context, the Lyee methodology has been described in recent papers [Nego 01a, Nego 01b] and has been used for a number of commercial software developments. This methodology proposes to largely skip the development step of creating the software design by providing a tool, called LyeeAll [LyeeAll], which leads from the description of the system requirements more or less automatically to the generation of executable code. The nature of the requirements description supported by this tool is quite different from the familiar concepts used in main-stream software engineering.

The purpose of this paper is to situate the Lyee methodology in respect to the main-stream software engineering approaches, concentrating on the notations used for describing system requirements. We focus on the concepts of requirement descriptions because this is the stage that is emphasized by the Lyee methodology, and it is also the stage from which the main-stream methodologies start. Since the area of requirements is still a field of active research in the software engineering commuity, we begin the discussion in this paper, in Section 2, with a personal review of what the important main-stream concepts for requirement specification are. In Section 3, we first explain that this paper is based on our limited knowledge of Lyee which is obtained from presentations and papers on the Lyee

methodology and reading examples of system specifications. The most pertinent references on the Lyee methodology are given. We start the comparison by commenting on the main concepts of the Lyee methodology as seen from the conceptual viewpoint of the main-stream software methodologies. Then, in Section 3.3, we present the main concepts of Lyee in more detail and relate them to main-stream software engineering concepts. Some important differences between Lyee and main-stream methodologies are pointed out in Section 3.4. Section 4 contains a discussion of certain questions that arise from this comparison, and is followed by the conclusions.

The discussions in Sections 3 and 4 refer at many places to the "Room Booking" example presented in [Sali 01]. We give an overview of this example in the Annex, and also present an alternate description of its requirements based on main-stream software engineering concepts.

We hope that the discussion and comparison presented in this paper, which represents the personal opinion of the author, could be useful in the further development of the Lyee methodology and its integration with other software development methods and tools.

2. Basic concepts for describing requirements and programs

The notations used to describe computer programs and the requirements that such programs should satisfy have developed over the last 5 decades from relatively low level notations to more abstract concepts, corresponding to the concern which concentrated first on how to write programs and later, when these programs became more complex, on how to specify the properties that the programs to be developed should satisfy. The purpose of this section is to give a (personal) overview of the most important concepts that have been used, and are still being used, when programs and their requirements are defined. Since these concepts have been found useful over the years by many practitioners, it is believed that they represent a good foundation to which the concepts of Lyee could be compared.

If we skip machine programming languages, since they are not oriented towards human understanding but ease of execution by the computer, we should first consider the concepts of high-level programming languages. A well-known book by N. Wirth was entitled "Algorithms + data structures = programs". This indicates that there are two aspects of programs: (1) the algorithmic aspect, also called control flow, which indicates the possible order of execution of certain basic program actions and (2) the data structure aspect which defines the data structures that are processed by the basic actions mentioned in the algorithmic part.

The main concepts for describing control flow are the following:

- Sequential execution.
- If and Case statements indicating under which conditions which subsequent actions should be executed.
- Goto statement indicating that the next action is out of sequence; this concepts is considered harmful for writing easily understood specifications, and the principles of Structured Programming indicate how one can write programs without using this construct. However, equivalent concepts also appear in higher-level design and requirement notations, such as the flow-chart-like notations for state transition diagrams and Petri nets (and the derived notations for the UML Activity Diagrams [UML] and Use Case Maps [Buhr 98].
- Structured loops (While, Repeat until, Loop forever).
- The procedure call mechanism including the passage of input and output parameters. This concept is very important, since it represents the main tool for defining

procedural (i.e. control flow) abstraction. An interesting feature, in this context, is recursion (not supported in all cases), which is important for processing recursive data structures, such as lists and trees.

The main concepts for describing data structures are the following:

- Basic data types, such as Integer, Boolean, Character, Bitstring or Octetstring.
- Data type constructors, such as Record (also called Struct), Discriminate Union, linear list (also called SequenceOf, or Array); including the recursive use of Record definitions allowing the definition of arbitrary graph structures, e.g. trees.
- Database schema constructors, including the above Record, but also the concept of a set of records (called Relation) and the concept of relationship between record types.

A third category of concepts is related to the architectural structure of software and hardware systems. They become essential for describing large systems consisting of many smaller components. The main concepts are the following:

- The system (to be constructed) and its environment.
- System components and their sub-components.
- Interfaces between components (and similarly between the system and its environment).

Historically, there were a number of important trends to make the development of computer programs more manageable. These trends went along with additional concepts of which the following appear to be the most important ones:

- Development of high-level programming languages (with the introduction of most of the concepts above).
- Structured programming with the aim to make the control structure of programs more easily understandable. One aspect was the avoidance of the Goto, another aspect was the relationship between data structures (introduced in Simula67 and Pascal) and corresponding control structures (which is another idea reflected in the book title "Algorithms + data structures = programs").
- Entity-relationship modeling of the data that is important for database applications [Chen 76].
- Data flow modeling (including its formalization as attributed (or colored) Petri nets).
- Information hiding, component interfaces, and object-orientation are aimed at a methodology for subdividing a system into components and sub-components. This goes hand in hand with defining the interfaces that a component provides for communication with the other components in the system. In the object-oriented paradigm, a component is called an "object" and the concept of "data type" is extended to the concept of "object class"; the interface of an object is only defined as far as its signature is concerned, but not concerning dynamic properties required for the output parameters.
- Pre-and Postconditions, specified in first-order logic (independently proposed in two papers by Floyd and Hoare in 1969) and included as debugging aid in the Eiffel language), allow the definition of the dynamic properties of procedure and function calls, as well as the dynamic properties of object interfaces.
- The concept of concurrency is important for operating systems and multi-user applications. It implies the consideration of interprocess communication and mutual exclusion for the access of shared resources. In the context of databases, it has led to the concept of a transaction, which is fail-safe and will be executed either completely (without interference of other concurrent transactions) or not at all.

Many of these concepts have been included in modern notations, for instance in the programming language Java and in the Unified Modeling Language (UML) [UML]. In the

context of this paper, we are mainly interested in the description of system requirements. UML includes a number of different notations intended for the description of system requirements, designs, and implementation structures. The UML Class Diagram provides a notation with a semantics very similar to traditional entity-relationship modeling. The Sequence and Collaboration Diagrams of UML are similar in nature to the Message Sequence Charts defined by the ITU [Z.120]. Also the UML State Diagrams have their correspondence in the SDL language [SDL]. All these notations are best suited for the description of system designs when one defines the behavior of the system components and their interworking. For describing system requirements, UML proposes the Use Case Diagrams and the Activity Diagrams. The former is very rudimentary; it defines the users and the use cases they may be involved in, but the properties of these use cases are simply explained in English. One may use an Activity Diagrams to describe the properties of a use case in a more formal manner. This notation has a semantics quite similar to Use Case Maps [Buhr 98], although the graphical notation is quite different. We think that the notation of Activity Diagrams or Use Case Maps are quite suitable for describing system requirements, including the dynamic properties. A related notation is also proposed in [Boch 00d]. Therefore we compare the Lyee methodology with the use of Activity Diagrams by considering a small example in the Annex. We note that we concentrate in the following on the semantic meaning of the concepts used in Lyee and Activity Diagrams, because we think that the graphic or textual representation of these concepts is of secondary nature.

3. Comparison of Lyee with conventional methods for describing requirements

3.1 What is the Lyee methodology ?

Before attempting a comparison of Lyee with conventional methods, one should have a good understanding of both, the Lyee methodology and the conventional methods. Section 2 tries to give an overview of the conventional methods. Concerning Lyee, I must say that I found it not easy to grasp the meaning of the Lyee concepts and notations. Among the various papers available in English, the following were most useful to my understanding. I found [Nego 01a] useful for the definition of the names of concepts used in Lyee. I found the best explanation of the meaning of the Signification Vector associated with the Pallets in [Nego 01b]. [Poli 02] talks about the same things in an easily understandable language, but remains relatively vague. I was therefore looking for a more concrete (operational) definition of the Lyee concepts, and some examples. I think that the case study of [Salinesi 01] is the most interesting example for this purpose. Note, however, that the concepts of Logical Unit and Word are called Defined and Item, respectively, which corresponds to names introduced in [Roll 01].

The paper [Roll 01] is interesting because it tries to define a meta-model for the concepts of the Lyee methodology (called concepts of the "Lyee software requirement layer"), and at the same time, the paper proposes a simpler meta-model (called "User requirements layer") which corresponds to a more abstract view of the Lyee concepts. In the subsequent comparison, we will refer to both of these layers. These two meta-models are shown in Figure 1.

It seems that the main argument for the promotion of the Lyee methodology is the statement that, using this methodology, the developer can easily define the requirements of the application and obtain the corresponding program largely automatically using the Lyee development tool [Lyee-All]. This means that the traditional software activities related to the subsequent development of the software design and implementation (including the related testing activities) would be minimized. Since the emphasis is put on the definition of the requirements (and this definition must still be made by the software expert in

conjunction with the user), we concentrate in our comparison on aspects related to the definition of the system requirements.

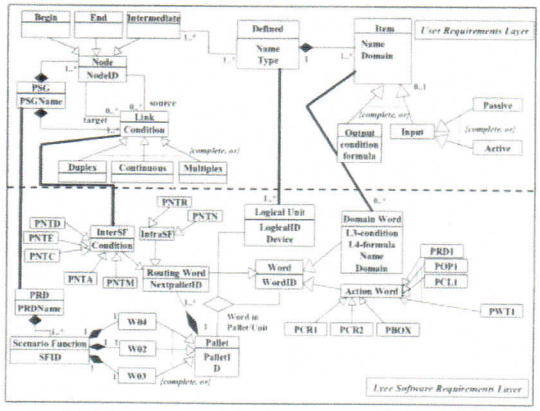


Figure 1 (from [Roll 01]): Meta-model of Lyee concepts and a more abstract view

Based on my current understanding of the Lyee methodology (which is certainly incomplete), I came up with the following comparison between the Lyee methodology and the traditional approach to requirements description. I hope that this discussion and comparison could be useful in the further development of the Lyee methodology and its integration with other software development methods and tools.

3.2. The Lyee methodology seen from the main-stream conceptual viewpoint

In Section 2, we identified three main categories of concepts for describing information processing systems, namely concepts for describing (a) control flow, (b) data structures and object classes and relationships, and (c) the architectural structure of software and hardware systems. The third category of concepts is of minor importance for the description of requirements, since requirements, by definition, describe the properties and behavior of a system as a whole, in terms of its interactions of the environment. The architectural structure of that system should normally be elaborated during the system design process, and not given as part of the requirements. Therefore we concentrate our attention in the following on the concepts used for describing (a) control flow and (b) data structures, objects and relationships.

It is in the area of control flow that much of the originality of Lyee lies. In fact, one may identify two three levels of control flow within the Lyee methodology. At the upper layer, there is the high-level control flow, expressed by the Process Route Diagrams (PRD). At this level, application-specific control flows are specified; this level has in fact much similarities with the main-stream activity diagrams, as shown in the example of the Annex (see also subsequent discussion in Section 3.3). At the next layer is the control flow within each Scenario Function. Here Lyee defines a standard control flow in the form of a loop (as seen for example in Figure 9 for SF1 and SF2, etc.). At the lowest level is the control flow within each of the three pallets of each Scenario Function. Here again, Lyee defines a standard control flow (as for instance depicted in Figure 3 of [Nego 01b]).

A major claim of the Lyee methodology appears to be the claim that this standard control flow at these lower two levels results in an important simplification of the requirements capture. This appears to be the major advantage of the Lyee methodology. It would be interesting to perform some controlled experiments in order to validate this claim.

In the area of data structures, objects and relationships, one can identify within the Lyee methodology again several "levels". At a higher level, we have application-specific

data structures, called Logical Units, which have some similiarities with object classes with attributes and method interfaces (as discussed in Section 3.3 in more detail). At a lower level, Lyee has on the one hand standard data type primitives, such as Integer, String, etc., like most programming and specification languages. On the other hand, Lyee includes some standard and rather complex data structures to support the processing of the "Words" (attributes) of Logical Units in relation with the standard control flow described above. Fortunately, in many cases, the latter data structure need not be visible to the person defining the requirements. It is my personal impression that more powerful concepts for describing object classes and relationships at the higher level would be useful for the Lyee methodology, as discussed in Section 3.4.

3.3. Correspondence between concepts

In the following, we list some Lyee concepts and discuss their meaning as compared with concepts familiar from the conventional methods for requirements description.

Domain Word: A Domain Word has an identifier, a name and a domain, which is a basic type (see last column in Figure 8). It is not clear what the difference between the identifier and the name is, since most examples use the same string for these two purposes. They represent input/output parameters related to a Logical Unit, and they also correspond to variables associated with the Pallets of Scenario Functions.

Logical Unit: A Logical Unit represents an interaction with the environment of the Lyee application. It has much similarity with an Remote Procedure Call (RPC) performed by the application accessing a component in the environment which is identified by the Type of the Logical Unit. For instance, as shown in Figure 10, the *CostomerDB* Logical Unit represents a query of the database getting the name of a customer given his identifier. Also a Logical Unit of Type *Screen* can be considered an RPC on the screen object which provides as output Words from the application (which become input parameters for the screen) the information to be displayed on the screen, and the response from the user is provided as input to the application.

If we consider that the components in the environment of the Lyee application are objects, then the Logical Units associated with a particular object can be considered to be defined methods on these objects. The output Words associated with a Logical Unit correspond to the input parameters of that method, and the output parameters of the method (we assume that there may be several, in contrast to popular languages such as Java) correspond to the input Words of the Logical Unit.

Scenario Function: As shown by the example in Figure 9, a Scenario Function contains three so-called Pallets, one for defining output (W04), one for receiving input (W02) and one for determining whether and where to continue the processing (W03). The semantics of the Scenario Function seems to be the very essence of Lyee. In simple situations, as for the example shown in Figure 9, the semantics corresponds to one or several method calls on objects in the environment and a subsequent choice of the next Scenario Function to be executed. As shown in the Lyee meta-model of Figure 1, the description of the possible next Scenario Functions and the conditions that are associated with this choice are contained in the Routing Words and the InterSF.Condition.

Process Route Diagram (PRD): While the low-level control structure of the Scenario Functions is a standard structure in Lyee, there are many application-dependent features

which are defined using a number of tables provided by the user interface of the LyeeAll tool. The Process Route Diagram is a convenient overview of many important properties of the Scenario Functions that are included in an application. As exemplified by Figure 9, it shows the overall control flow of the application and may include information about the interactions with the environment and informal comments about the conditions for the control flow choices. This information corresponds largely to the information provided in a UML Activity Diagram, as shown in Figure 7, for example.

3.4. Differences

Besides the corresponding concepts discussed above, there are a number of important differences, such as the following.

Flat aggregation hierarchy: The Words included in a Logical Unit as input or output parameter are of basic type, it is not possible to consider Words that are objects with attributes, for instance. In the object-oriented approach, an attribute of an object may be another object, and so on. This is very useful for describing complex aggregation relationships. This is not possible in Lyee.

No modeling of relationships: Lyee provides no tools for modeling relationships between object classes, as used in entity-relationship modeling and UML Class Diagrams.

Abstraction: In Lyee, there seems to be no way of introducing procedural abstraction; there is no procedure call mechanism with parameter passing.

Relying on the underlying implementation language: For describing conditions for the choice of control flow or for the validity of input received, Lyee has no own notation, but relies on the notation of the programming language into which the Lyee specification is translated when an implementation is generated. The same holds for the description of the expressions that define output values.

Arrays: The MBA Database case study [MBA_01] includes the use of an Array concept in Lyee. However, it is not clear how this can be used. It seems that for each Domain Word that is declared to be an array, there must be another Domain Word that indicates the length of the array. Different Domain Words of type array may form the rows of a table in the user interface (where each array index corresponds to a table row). Data processing seems to be defined only within each row (without interference between different array index values; that is, A[I] := B[I+1] is not allowed).

4. Discussion of further issues and questions

Although there exist some correspondence between certain concepts of Lyee and the traditional concepts of programming languages and requirement descriptions, it is clear that the Lyee methodology for software development is quite different from current main-stream approaches. Some further issues and questions are identified in the following.

Lyee's application domain: Most applications of Lyee that I have seen involve a database and users performing read and/or update transactions on the database. An important aspect is the graphical user interface, the validation of input data and the presentation of the

appropriate results. It is not clear how much the Lyee methodology is suitable for other kinds of application domains. The following features of Lyee may limit its usability in certain areas: (1) There are no recursive procedures which makes the description of the processing of complex data structures, such as trees, very difficult. (2) There is not much support for the decomposition of a system into separate components (well, you may notice that this is a software design issue, and not related to requirements); this makes the construction of distributed systems with Lyee more difficult. Therefore we have the following question: What is the intended domain of application of the Lyee methodology?

Boundary software: Once the requirements of a system to be developed have been determined and entered into the LyeeAll tool, this tool will automatically generate implementation code for the application. However, this generated code requires so-called boundary software, which is written by hand, for realizing the interactions with the components in the environment (such as screen, database, etc.). In some sense, this includes also the software for creating the graphical user interface (for which other main-stream software tools are used). The question then is: How important is the effort to create the boundary software ? – Again, it seems that the case of distributed applications has a disadvantage, since it involves communication between several components, of which one or several may be created using the Lyee methodology.

The declarative nature of Lyee: One of the stated advantages of the Lyee methodology is the fact that the lower details of the control flow, that is, the one within each Scenario Function, has a standard form (see for instance Figure 3 in [Nego01b]). This makes the understanding of the programs easier. Through the recursive loop in each Scenario Function, and similar embedded loops within each Pallet, the nature of a Scenario Function appears to be like a mathematical function which determines its output from the state of variables set by the execution of previous Scenario Functions and the own input received. It also includes the checking of input validity and processing of exceptional situations (which, however, is not automatic, but must be specified by the designer, unless the default processing is desired). Let us consider some specific examples:

- Pallet recursion: Let us assume that the values of three output Words B, C and D depend on the value of an input Word A, and are defined by the following assignment statements (which are part of the W04 Pallet and must be written in the associated implementation language) " B := A; C := D + 1; D := B *2; ". Since during the first execution of the W04 Pallet, the value of D is not defined when it is required for the calculation of the value of C, the latter value will remain undefined and the Pallet is executed a second time during which the yet undefined values will be determined (if possible).
- Circular definition: In the case of circular definition, e.g. " B := A; C := D + 1; D := C *2; " the Words involved in the circularity will remain undefined. Question: For what is Pallet recursion useful ?
- Programming loops: It is possible to use the inherent recursive repetition of a Scenario Function (for instance in Figure 9, the loop from the exit of Pallet W03 in SF01 back to W04, the beginning of this Scenario Function). For example, a loop that could be written in a programming language as "A := input; loop while A < 1000 { A := A * 2; } " could be realized in Lyee notation [Arai 02] by two Scenario Functions, one reading the initial value of A and the other looping until the termination condition is satisfied (this checking would be done by the W03 Pallet). Question: Is this a natural example ? - It seems to me that the Lyee methodology was not developed for writing such examples. But what should one do in cases where the requirements say: "Find the largest power of A that is smaller than 1000"?

Meaningful identifiers: We teach our students to use meaningful identifiers in programs and requirements definitions. All Lyee system specifications use numeric identifiers for Logical Units (e.g. screen1, screen2, etc.), Scenario Functions (e.g. SF1, SF2, etc.) and for most other things in Lyee specifications. The only exception are the Word identifiers that are often meaningful. Question: Are these numerical identifiers just bad practice, or does the LyeeAll tool limit the form of identifiers that can be used, or is there a good reason for using numerical identifiers ?

Complex conditions or data manipulations: In the example of the Room Booking example in the Annex, the definition of the database queries (in terms of an SQL query) was left to the boundary software, which means that it was realized outside the Lyee framework. Question: How could complex conditions or complex data processing algorithms be described within the Lyee framework ? – For instance, could one use the Lyee methodology to define a program fragment that realizes the following user query in the context of the MBA Database Application System [MBA 01] which includes the course offerings and the course schedules of students: "Please, display all courses that have no conflict with my current course schedule".

5. Conclusions

In conclusion, we can say that the Lyee methodology is an intriguing approach to software development and it is difficult to understand from the perspective of the current main-stream software engineering paradigms. This paper makes an attempt at comparing the concepts that underlie the Lyee methodology with the concepts that appear to be important for the description of software system requirements within the main-stream software engineering methods. While we identified several relations between concepts in Lyee and corresponding main-stream concepts, we also pointed out a number of important differences, and we leave the reader with a number of questions which, we think, merit further discussion.

Acknowledgements

I would like to thank Fumio Negoro and his team for many interesting discussions and my initial introduction to the Lyee methodology. I also would like to thank Hamid Fujita for encouraging me to work on this topic and for many fruitful discussions. Finally, I would like to thank Carine Souveyet (Paris) and Osamu Arai (Catena, Tokyo) for answering some of my more detailed questions.

References

[Arai 02] O. Arai, personal communication, June 2002.

[Boch 00d] G. v. Bochmann, Activity Nets: A UML profile for modeling work flow architectures, Technical Report, University of Ottawa, Oct. 2000.

[Buhr 98] R.J.A.Buhr, Use Case Maps as architectural entities for complex systems, IEEE Tr. SE, Vol. 24 (12), pp. 1131-1155, 1998.

[Chen 76] P. P. Chen, The Entity-Relationship model - Toward a unified view of data, ACM Trans. on Database Systems, Vol. 1, No. 1, March 1976, pp.9-36.

[LyeeAll] Software development tool built by Catena Corp. and the Institute of Computer Based Software Methodology and Technology.

[Nego 01a] F. Negoro, Intent operationalisation for source code generation, in Proc. SCI 2001, Orlando, Florida, USA, July 2001.
[Nego 01b] F. Negoro, A proposal for requirement engineering, in Proc. ADBIS-2001, Sept. 2001, Vilnius, Lithuania.
[MBA 01] The Institute of Computer Based Software Methodology and Technology, MBA database application system by using Lyee methodology, Internal Report, April 2001.
[Poli 02] R. Poli, Automatic generation of programs: An overview of Lyee methodology, Draft report, university of Trento (Italy) and Mitteleuropa Foundation, March 2002.
[Roll 01] C. Rolland and M. Ben Ayed, Meta-modelling the Lyee software requirements, in Proc. of an intern. conference, pp. 95 – 103.
[Sali 01] C. Salinesi, M. Ben Ayed, S. Nurcan, Development using Lyee: A case study with LyeeAll, Internal Technical Report TR1-2, University of Paris I and ICBSMT, Oct. 2001.
[SDL] ITU-T, Rec. Z.100: System Description Language (SDL), Geneva, 1999.
[UML] OMG, Unified Modeling Language Specification, Version 1.4, Mai 2001, http://www.omg.org
[Z.120] ITU-T, Rec. Z.120: Message Sequence Chart (MSC), Geneva, 1999.

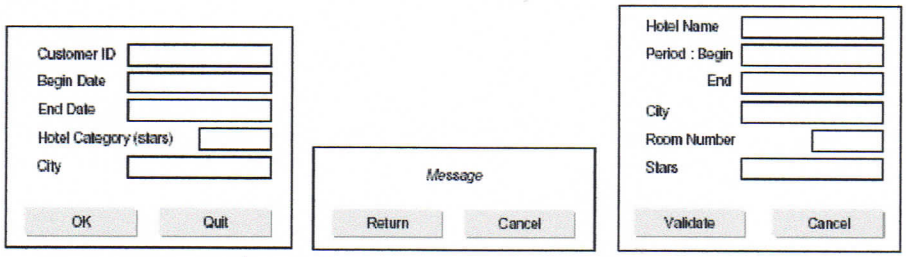
Annex: The Room Booking Case Study

The Room Booking Case Study is described in [Sali 01]. It describes the use of the Lyee methodology for the development of a program which provides a service for booking hotel rooms. The program has a graphical user interface. In the following, we give an overview of the requirements for this room booking system based on the documentation in [Sali 01], and provide an Activity Diagram and related Class Diagrams which describe the same requirements in UML.

A.1. Room Booking requirements described in UML

A.1.1. Informal description of the system requirements

The following is a summary of [Sali 01], Section 1. The Room Booking application uses a database that contains information about hotel rooms available in different cities. It also interfaces with the user and lets a user select a room for a certain period and make the reservation. A sketch of the user interface is provided in Figures 2 through 4. Figure 2 shows the initial screen where all the defined fields are input by the user. After pushing the "OK" button, the user sees either the screen of Figure 3 (where the *Message* may either read "The customer xxx does not exist in the database" or "No room is available for yyy" and xxx is the customer identifier and yyy is the customer name) or the screen of Figure 4, where all fields are output by the system. The user may then confirm the reservation by pushing the "Validate" button, or cancel the reservation and come back to the initial screen. The schema of the database that is used is given in Figure 5.



Figures 2, 3 and 4 (from [Sali 01]): (1) Initial screen, (2) message window, (3) validation

Hotel (HotelID, HotelName, Stars, City).

Room (RoomID, HotelID)
Customer (CustomerID, CustomerName)
RoomAvailability (RoomID, HotelID, BeginDate, EndDate)
Booking (BookID, HotelID, RoomID, CustomerID, BeginDate, EndDate)

Figure 5: Schema of the Room Booking database (from [Sali 01])

A.1.2. Relevant Class Diagrams

The logical structure of the user input described in Figure 2 and of the system output described in Figure 4 can be translated in a straightforward manner in corresponding UML Class diagrams. For instance, a class diagram for the output to the user related to the screen of Figure 4 is shown in Figure 6(a). Similarly, a Class Diagram including relationships can be developed to describe the database schema, as shown in Figure 6(b).

We note that the layout of the user interface, as shown in Figures 2 through 4, can not be described in UML. By the way, the situation is similar with the Lyee methodology. One assumes that the user interface layout is separately described. And suitable tools exist for generating automatically programs (in Visual Basic or Java, or as HTML pages) that realize the given interface layouts.

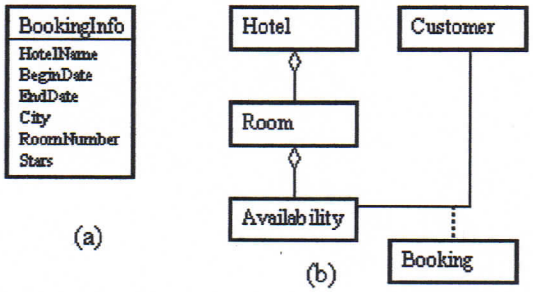


Figure 6: (a) The BookingInfo class corresponding to the output of the screen of Figure 4; (b) Class diagram representing the database schema of Figure 5 (without the class attributes)

A.1.3. Dynamic behavior requirements

The dynamic behavior of the Room Booking system can be described by the Activity Diagram shown in Figure 7. The activity *Prepare booking request* displays first the initial screen of Figure 2 and lets the user fill in the fields. The four subsequent actions correspond to the following four cases:

- The user pushes the *Quit* button.
- The customer identifier is not in the database.
- No suitable room is available.
- A room is available and may be reserved.

Case (4) leads to the activity *Confirm booking*, which displays the screen of Figure 4 and lets the user validate the reservation or go back to the initial screen.

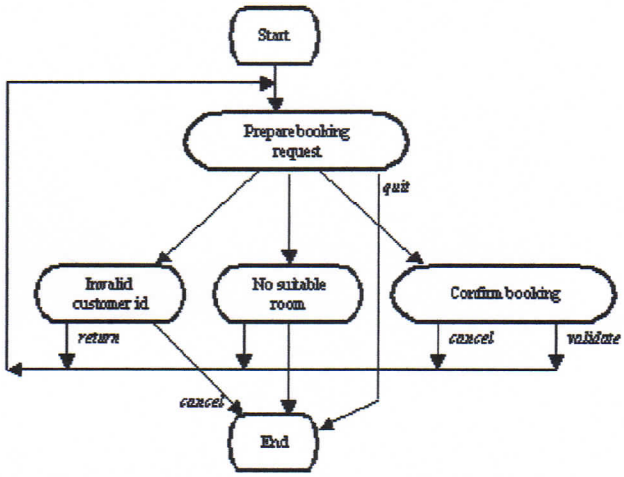


Figure 7: Activity diagram showing requirements on the dynamic behavior

We note that, in this example, each activity has the same sequence of interactions with the user: First a screen is presented to the user including values in certain information fields, and then the user may enter date into certain fields and pushes one of the buttons.

We also note that this activity diagram describes the behavior of the whole Room Booking system (including the database). In the case study of [Sali 01], the Lyee methodology is applied to specifying the Room Booking application program, which interacts with the database through SQL queries. The application, therefore, has one additional interface, namely the one for querying the database. If one wants to define the requirements for this application program, one has first to do some analysis in order to determine the type of SQL queries that are appropriate for this application. An activity diagram describing the dynamic requirements of the Room Booking application could then be obtained from the activity diagram of Figure 7 by refining the activities within the diagram and including the appropriate database queries and answers.

A.2. Development of the Room Booking example with Lyee methodology

In this subsection, we try to summaries the main steps of the case study described in [Sali 01]. The specification of the Room Booking application proceeds in two steps: first the aspects of data structures and interfaces and then the aspect of dynamic behavior (control flow). The first aspect is essentially covered by the definition of the so-called Logical Units. A Logical Unit corresponds to a screen or a database query and always includes input and output attributes. The Logical Unit and their attributes (called Words) for this application are shown in the table of Figure 8. Note that the last table column indicates the basic datatype of the Words (9 = numeric, X = alphanumeric, K = button); the input /output nature of the Words is not indicated in this table.

	ID	Name	Domain
For screen1	CustomerID	CustomerID	9
	BeginDate	BeginDate	X
	EndDate	EndDate	X
	Stars	Stars	9
	City	City	X
	CmdOK	CmdOK	K
	CmdQuit	CmdQuit	K
For screen2	HotelName	HotelName	X
	Begin	Begin	X
	End	End	X
	City	City	X
	RoomNum	RoomNum	X
	Stars	Stars	9
	CmdValid	CmdValid	K
	CmdCancel	CmdCancel	K
For screen3	Message1	Message1	X
	CmdReturn	CmdReturn	K
	CmdCancel	CmdCancel	K
For screen4	Message2	Message2	X
	CmdReturn	CmdReturn	K
	CmdCancel	CmdCancel	K
For CustomerDB	CustomerID	CustomerID	9
	CustomerName	CustomerName	X
For RoomAvailability	RoomID	RoomID	X
	HotelID	HotelID	X
	BeginDate	BeginDate	X
	EndDate	EndDate	X
For BookingDB	BookID	BookID	9
	RoomID	RoomID	X
	HotelID	HotelID	X
	CustomerID	CustomerID	9
	BeginDate	BeginDate	X
	EndDate	EndDate	X
For.AvailableRoom	HotelName	HotelName	X
	HotelID	HotelID	X
	RoomID	RoomID	X

Figure 8 (from [Sali 01]): List of defined Domain Words

The dynamic aspects of the requirements are defined in the form of one of several so-called Process Route Diagrams (PRD), which are graphical representations of the control flow aspects of the application program. The PRD for the Room Booking application is shown in Figure 9.

One basic characteristic feature of the Lyee methodology is the standardized control structure which includes an internal loop within each of the W04, W02 and W03 Pallets and the loop repeating the execution of the Pallets within a given Scenario Function in the order W04, W02 and then W03 until no change occurs to the values of the variables associated with these Pallets. The PRD of Figure 9 includes 6 such Scenario Functions, named SF01 through SF06. From an abstract point of view, the PRD of Figure 9 defines the same control flow structure as the activity diagram of Figure 7; the Scenario Functions SF01 and SF02 correspond to the activity *Prepare booking request* , and SF03 and SF06 correspond to the *Confirm booking* activity. The other Scenario Functions SF04 and SF05 correspond to the other two activities shown in Figure 7. It is to be noted that the PRD, in addition, shows information about what items are input or output and the interactions with the database (since the application has an explicit interface with the database).

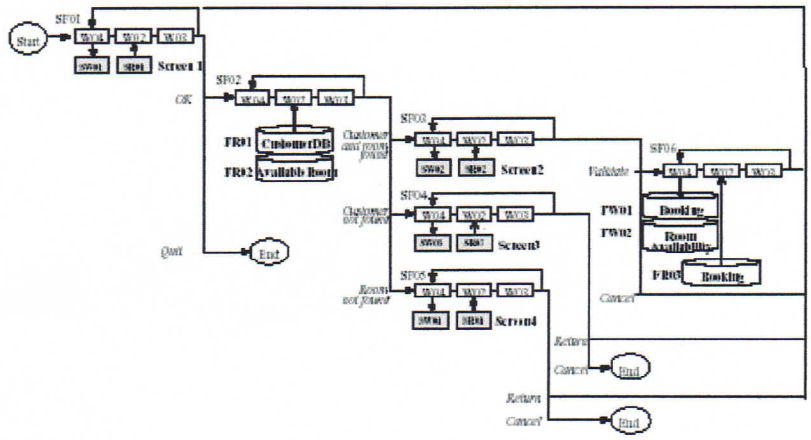


Figure 9 (from [Sali 01]): Process Route Diagram of the Room Booking example

ID	Name	Type
Screen1	Hotel Reservation	Screen
Screen2	Reservation	Screen
Screen3	NotAvailableRoom	Screen
Screen4	CustomerError	Screen
CustomerDB	Customer	File
AvailableRoomDB	AvailableRoom	File
RoomAvailabilityDB	RoomAvailability	File
BookingDB	Booking	File
Key_CustomerDB	Key_Customer	KeyWord
Key_AvailableRoomDB	Key_AvailableRoom	KeyWord
Key_RoomAvailabilityDB	Key_RoomAvailability	KeyWord

Figure 10 (from [Sali 01]): List of Logical Units