Designing Logical Architectures of Software Systems

Sungwon Kang, Yoonseok Choi
School of Engineering, Information and Communications University, Korea
{kangsw, yschoi}@icu.ac.kr

Abstract

It is commonly agreed that architecture design is essential for development of complicated software and that depending on viewpoints software architecture can be captured in various views. Among them, the logical view is the earliest and foremost view that software developers should consider because it shapes the later phase of architecture design and the subsequent development. In this paper, we present an architecture description language for logical architecture and its associated design method. The main characteristic of the language and the design method are that they support recursive top-down design so that logical architectures for software systems of any scale and complexity can be designed. We apply them to a realistic system and show their efficacy.

1. Introduction

As the importance of software architecture in software development is increasingly recognized, researchers increasingly contributed to software architecture research. One of the results they found is that software architecture cannot be adequately captured in a single view but needs to be captured from various viewpoints. Such viewpoints include logical view, runtime view, module view and deployment view and others [1][7][8]. Then depending on the specific view taken, each instance of software architecture obtained is respectively a logical architecture or a runtime architecture or a module view architecture and so on. Among various views' architectures, the logical architecture is the one that should be established before any other architecture so that it guides derivation of the other views' architectures and shapes the whole subsequent development [7][8].

However, research on logical view of software architecture is not well established yet. So, for example, Bass et al did not include the logical view among the set of views they considered [1]. Kruchten recognized the importance of the logical view but the granularity of architecture design elements in his work was too small since they were classes and objects rather than subsystems and components [8]. Hofmeister et al did not provide an appropriate level of abstraction for logical architecture by essentially considering runtime architecture instead [7].

For all views of software architecture, there are two important related questions. One question is, "What is the adequate Architecture Description Language (ADL) for the view? Specifically, what are the architecture design elements for the architecture and how can their relationships be expressed?" The other question is, "Given such an ADL, how can we actually carry out architecture design and use the ADL to express the design results?" This paper addresses these two questions for logical architecture.

The remainder of the paper is organized as follows: In Section 2, related works are surveyed and their limitations are discussed. Section 3 provides the foundation for our work. For that, starting with the definition and the goals of logical architecture, we discuss what needs to be done for logical architecture design. Section 4 presents an ADL for logical architecture called ALFReD. Section 5 presents a logical architecture design procedure called ReDLArch. In Section 6, the efficacy of ALFReD and ReDLArch are demonstrated by working out a logical architecture design with an example. Section 7 concludes the paper.

2. Related Works

In this section, we survey related works and discuss their limitations. For that, first we surveys three widely known software architecture frameworks. Among the three frameworks, only Kruchten's 4+1 Views framework recognizes and provides a due role to logical view. But we see that the design elements of the Kruchten's logical view are too fine-grained and its design method does not allow recursive design. Lastly, we look at the existing ADLs and points out that they focus on runtime architecture rather than logical architecture.

2.1 Various Software Architecture Frameworks

Researchers on software architecture proposed various software architecture frameworks, each consisting of a set of architecture views. There are three well-known frameworks proposed by Kruchten [8], Hofmeister et al [7] and Bass et al [1][5]. In this paper, we will call them respectively *Kruchten's 4+1 Views*, *Sie-*

mens Four Views and *SEI Views*. The Kruchten's 4+1 Views framework considers five views, each of which addresses a specific set of concerns. In the framework, the architectural design decisions are captured in four views, i.e. the logical view, the development view, the process view and the physical view and the fifth view is the use cases view. The use case view consists of a few selected use case scenarios to drive the discovery of the architectural elements during the architecture design and to validate and illustrate the architecture design later. Table 1 compares the views of this paper with the Kruchten's 4+1 Views framework and the other two frameworks. Although the exact meanings of various views differ from framework to framework, we regarded them as the same views if the intended concepts are similar.

Table 1. Comparison of architectural frameworks

Views of this paper	Kruchten's 4+1 Views	Siemens Four Views	SEI Views
Logical View	Logical View		
Code View	Development View	Code View	
Module View		Module View	Module View
Deployment View	Physical View	Execution View	Allocation View
Runtime View	Process View	Conceptual View	Component & Connector (C&C) View
	Scenario View		

2.2 Kruchten's Logical Architecture and Logical Architecture Design

In the paper [8], Kruchten said, "The logical architecture primarily supports the functional requirements—what the system should provide in terms of services to its users. The system is decomposed into a set of key abstractions, taken (mostly) from the problem domain, in the form of *objects* or *object classes*." So in the Kruchten's logical view, the main concern of logical architecture is functionality distribution and the design elements for logical architecture are classified into components and connectors, where a *component* can be class, class utility, parameterized class, and class category and a *connector* can be a relationship between components such as containment, aggregation, usage, inheritance and instantiation. Therefore we can see that Kruchten's logical architecture focuses on comparatively low-level design elements. Also it's connector concept contrasts with that of Siemens Four Views and SEI Views in that in the latter frameworks connector means a special component that connects components. Kruchten's architecture design method targets all views of architecture at once and no specific design method is given for logical architecture.

2.3 ADLs

In the past, many ADLs were developed with a view to help analysis, design and evolution by formalizing software structure description notations [4][11]. Some representative ADLs are Wright [6], UniCon [13], Rapide [9] and C2 [11]. Wright focuses on modeling and analyzing the behavior of concurrent systems. UniCon focuses on generating the glue code for the existing components to make them work smoothly by using a Common Interaction Protocol. C2 focuses on representing distributed and real-time systems. ACME is an architectural description interchange language that allows various ADLs to be interchanged with each other [6].

Since the existing ADLs did not explicitly consider multiple views of software architecture and primarily focused on the runtime view, they are not suitable for logical architecture description. For example, the previous ADLs included for runtime view design elements such as port, role, component instance and connector instance. But these are the concerns for which the decisions can be deferred to the later architecture design stage or to the detail design stage.

3. The Logical Architecture and Its Design

In this section, we discuss what logical architecture is and how its design should be carried out. For this, we first give our definition of logical architecture. Then, based on the definition, the goals of logical architecture design are identified as buildability and reusability. Finally, we discuss what the design process for logical architecture should be like to achieve those goals.

3.1 The Definition of Logical Architecture

The intended meaning of the term *logical architecture* often appears clear but researchers differ when it comes to the details of what it means. In the paper [8], Kruchten said, "The logical architecture primarily supports the functional requirements – what the system should provide in terms of services to its users. The system is decomposed into a set of key abstractions, taken (mostly) from the problem domain, in the form of objects or object classes." Hofmeister et al talked about conceptual architecture rather than logical architecture in their work [7]. But their conceptual architecture was actually closer to runtime architecture as they say, "In the conceptual view, you model your product as a col-

lection of decomposable, interconnected conceptual components and connectors" and their components and connectors are actually runtime elements.

The definition of logical architecture in this paper is as follows:

Definition A *logical architecture* for a software system defines a collection of components and their relationships so that realizations of the components and their relationships can satisfy all the requirements of the system.

Logical architecture can be descriptive or prescriptive. Descriptive software architecture is the architecture of the existing system and prescriptive architecture is the specification of the architecture for the software system to be built. The above is the prescriptive definition of logical architecture. Here the term *component* means an entity that resides inside an enclosing entity and constitutes part of it. So a component can be a target for logical architecture design unless it is an atomic component, which is a component that we decide not to decompose any further. So the above definition does not prevent the design process for logical architecture from proceeding in a recursive manner.

3.2 The Goals of Logical Architecture Design

Given the notion of logical architecture as defined in Section 3.1, what are the ultimate goals that should be pursued to have a "good" logical architecture design? The answer depends on what one regards as the most important things in architecture design. We believe that as the first architecture design stage we should abstract from less important concerns as much as possible in logical architecture design and focus only on the most important concerns. We assert that the most important concerns for logical architecture are (1) how we can successfully build a functionally working system at all and (2) how we can save development effort as much as possible. For the first concern, what needs to be done is to reduce the complexity of software development. This can be naturally done by decomposing the task of developing a system into more manageable tasks of developing smaller subsystems (i.e. components) and putting them together. As a general problem solving strategy, it is often referred to as *the divide-and-conquer strategy*. The second concern can be addressed to a great extent by not duplicating work during development or by reusing what can be reused as much as possible. We call these two concerns *buildability* and *reusability* respectively and claims the following proposition.

Proposition The goals of logical architecture design are to assure buildability and reusability.

3.3 Logical Architecture Design Process

Buildability and reusability concerns are different from other concerns about software systems in that regardless of the customer's requirements they are applied by the developers for the sake of development. After the logical architecture is designed, the other concerns from the customer can be incorporated at other phases of development.

We view logical architecture design process as a process that determines the structure of the system under construction and specifies the constituent components that participate in the structure. We can do this by repeating (1) the step of partitioning or factoring, followed by (2) the step of specifying components by assigning responsibilities to the components and defining the interfaces between them. This process is depicted in Figure 1.

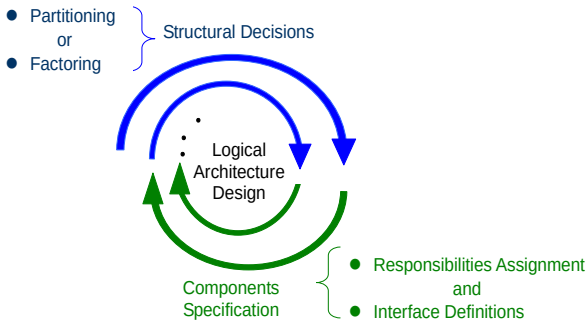

Figure 1. Design process for logical architecture

In the first stage of Structure Decisions in Figure 1, we find out a set of components and a structure consisting of them from the requirements for the whole system. Each component would have an associated set of requirements and the components together would satisfy the requirements for the whole system. This step can be either a *partitioning* step or a *factoring* step. If it is a partitioning step, then the main concern is how to make the system easy-to-build by applying the divide-and-conquer strategy. If it is a factoring step, then the main concern is to maximize reusability by factoring out a certain functionality that has already been realized by an existing component or by factoring out a common functionality from the existing components.

In the second step for components specification, new but less complicated requirements than the initial system's requirements are derived and assigned to the components. In this step, not only functional requirements but also non-functional requirements as well as constraints are distributed to components. Also well-specified components should define provider-user protocols for interactions with other components. Such specification includes not only how to use services but

also the context of the services as well. These two steps are repeated until there is no need for introducing further structure.

4. ADL for Logical Architecture

To meet the concerns of logical architecture, the ADL for logical architecture should have the following characteristics: (1) the language constructs should be the logical architecture design elements, (2) it should allow for a hierarchical manner description to help top-down design and, furthermore, (3) description notation for each layer of the hierarchy should be the same. The second and the third characteristics would help the top-down recursive design description and design process. This section presents an Architecture description Language For Recursive Design (ALFReD) that has all the characteristics stated above.

Traditionally, boxes and lines as shown in Figure 2 were used to represent structures of systems. Boxes represent computing entities and lines represent the relationships between components. We call them *logical component* and *logical relationship* to emphasize that they are design elements for logical architecture.

Figure 2. Box and Line

Logical relationships can be elaborated with the same decorations that are used for association between classes in UML [3] such as roles and multiplicities as illustrated in Figure 3.

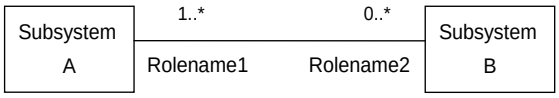

Figure 3. The System represented with the Consumer-side and the Producer-side

Figure 4 shows a system represented with one box. The internal architecture may be undetermined yet or may be determined but not shown because only the topmost level view is given. In the first case, it has all the requirements associated with it. In the latter case, it may have the internal architecture design as shown in Figure 5. In Figure 5, there are two logical components and they have a certain logical relationship between them. The nature of the relationship can be expressed with decorations of the association and further through separate sets of requirements for the two components.

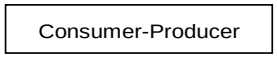

Figure 4. The system represented with one box

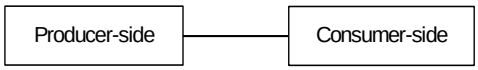

Figure 5. The System represented with the Consumer-Side and the Producer-Side

In ALFReD, a component is described with a quadruple consisting of *name*, *observable requirements*, *non-observable requirements* and *constraints*. The Figure 6 shows how we describe a logical component in ALFReD.

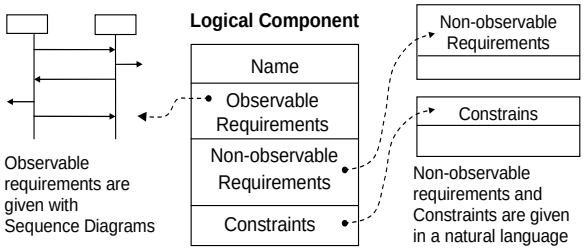

Figure 6. ALFReD specification of a logical component

ALFReD uses sequence diagram to describe component's functional and non-functional requirements and interfaces that can be expressed with interactions with other systems or components. With the notation for condition and repetition, sequence diagram is expressive enough to describe most computation and interaction behavior. The further advantages of sequence diagram are that it is a formal language such that use case scenarios can be easily translated into it and functionality given in sequence diagram for a system can be semi-automatically distributed to their components.

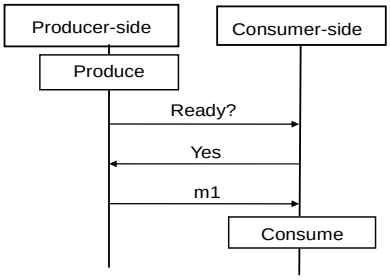

Figure 7. A sequence diagram for component interaction

Suppose that from the system's requirements we have decided that the system should consist of Producer-side component and Consumer-side component and control flow should exist between them. When we look closely into Producer-side and Consumer-side for further design, they may have the internal behaviors that can be derived from the interaction in Figure 7. One possible scenario for such interactions is depicted in Figure 8. Then the interactions between the internal components within of the Producer-side component and

the Consumer-side component can in turn be described in sequence diagrams. They are indicated with the shaded areas in Figure 8.

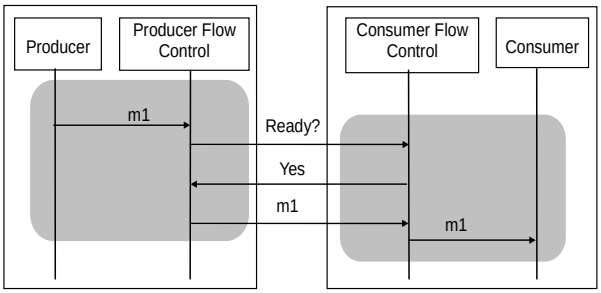

Figure 8. Refinement of Figure 5

5. Design Method for Logical Architecture

We argued in Section 3 that buildability and reusability are the most important concerns that should be considered for design of logical architecture. In Section 4, we presented a recursive design language for logical architecture. In this section, we show how logical architecture can be designed using the ALFReD language in such a way that the two concerns are properly addressed. For this, we first consider the following two questions: (1) what actions need to be taken to achieve those goals and (2) what guidelines need to be followed in taking the actions so that we get the best results by performing the actions. Table 2 summarizes the answers to these questions.

Table 2. The process to achieve the goals of logical architecture

Buildability	Purpose	Decompose the system into components to make it realizable and implementable.
	Actions	Partition the system based on architecture patterns. Partition the system into collaborating design elements by considering decoupling and cohesion.
	Guidelines	Ensure understandability. Consider developer's skill. Consider the available number of developers.
Reusability	Purpose	Reuse components as much as possible to make development efficient.
	Actions	Factor out part of functionality to an already existing component. Factor common functionality of two components to a separate component.
	Guidelines	Consider trade-off between amount of reuse and amount of effort for reuse. Consider extending functionality of a component beyond what is required so that the existing component can be reused.

Architectural patterns are well-known design solutions for commonly arising architectural design problems. Architectural patterns can play an important role in decomposing a component into the simpler collaborating components because by adopting the existing proven solutions into the design they promote buildability. Each architecture pattern has its name, the requirements it addresses and the solution design. Table 3 shows a list of names for some architectural patterns that can be used in the logical architecture design.

Table 3. Architectural patterns for buildability

MVC pattern	Presentation-Abstraction-Control pattern
Layered pattern	Strategy pattern
Whole-Part pattern	Process Control pattern
Pipes and Filter pattern	Event-Based pattern
Blackboard pattern	

Our procedure for logical architecture design is described in Figure 9. We call it the *ReDLArch* method for Recursive Design for Logical Architecture method. Our design method consists of the step for buildability and then the step for reusability. The buildability step first starts with the sub-step that utilizes the existing architecture patterns and then performs the design not relying on architectural patterns. This second sub-step of buildability performs *divide-and-conquer* based on the principle of "decoupling and cohesion" such that individual components have maximum cohesion in themselves and minimum coupling exists between them. This will make it much simpler to perform the next level of design of the newly introduced components. The final phase of design explores reusability possibilities and performs two kinds of factoring to the existing components. One is factoring into an existing component and the other is factoring out from the existing components.

Each of these three steps is immediately followed by the step of fleshing out the newly introduced components. Through fleshing out, each component becomes a system that is complete with all the pieces necessary for further design. When we design logical architecture, we not only make decisions on structure among components but also always distribute the requirements that have not been taken care of at the current level iteration to the appropriate components. This enables the next level iteration of the procedure to continue afresh only with the resulting components and makes the design procedure top-down and recursive.

Note that for the first iteration of the application of the procedure of Figure 9, the input system is the whole system. Also note that the method can be customized in various ways. One dimension is whether to base structure decisions on a complete set of requirements or on a selected set of more significant quality attributes and constraints.

ReDLArch - Logical Architecture Design Procedure
Input: A system *S* specified with ALFReD
Output: A logical architecture specified with ALFReD

if *S is simple* **then** *return ;* /* Else *S* requires design */
/* Perform buildability design */
if *there are architectural patterns applicable*
then *introduce components for the architectural patterns;*
Flesh out the components for the architectural patterns;
if *there are remaining requirements for S not taken of by the architectural patterns*
then *introduce new components for the remaining requirements by considering decoupling and cohesion;*
Flesh out the new components by distributing the remaining requirements for S to the new components;
/* Perform reusability design */
if *there are existing components for part of the system functionality or there are commonalities among the components introduced so far*
then *factor them out as separate components;*
Flesh out the components that are affected in this step by re distributing the relevant requirements ;
Recursively apply this method to each component;

Figure 9. Logical architecture design procedure

6. An Application Example

In this section, we show how ALFReD and ReDLArch can be used for logical architecture design. The application target is a simplified version of the Point Of Sale (POS) system that appears in [9].

6.1 The POS System

The requirements are as follows:
(1) When customers bring goods for check-out, cashier handles a sale process through POS.
(2) Customers can return damaged goods.
(3) Product manager can manage goods.
(4) Sale manager can see the summary of the sale information and profit for specified period.

Figure 10 is the use case context diagram. Figure 11 describes use case scenarios for the Process Sale use case. Figure 12 shows the system constraints. The use case scenarios for Handle Returns (HR) use case are omitted.

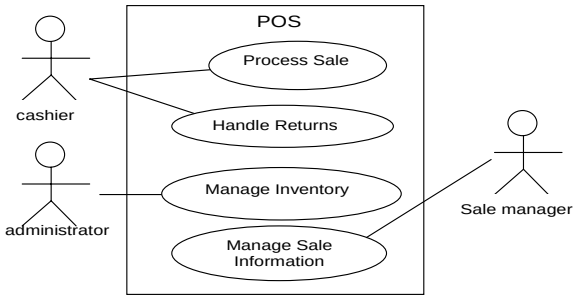

Figure 10. Use case context diagram for the POS system

Process Sale Use Case Scenario

Pre-condition: Cashier is identified and authenticated.
Post-condition: Sale is successfully saved

Main Success Scenario
1. Customer arrives at POS checkout with goods
2. Cashier starts a new sale
3. Cashier enters item identifier
4. System records sale line item and presents item description, price, and running total

Cashier repeats Steps 3-4 until all the items are identified
5. Casher tells the total price to the customer
6. Customer pays expense
7. System handles payment and logs complete sale and then returns a receipt
8.

Alternative Flow for 3~5
(3-4) Customer asks Cashier to remove an item from the purchase
1. Cashier enters item identifier for removal from sale
2. System displays updated running total

Figure 11. Process Sale (PS) use case scenario

Constraints
C1. Different types of users need different user interfaces.
C2. Maximum of 20 cashiers can use the system together.

Figure 12. Constraints for the POS system

Figure 13 shows the system interaction diagrams derived from Process Sale use case.

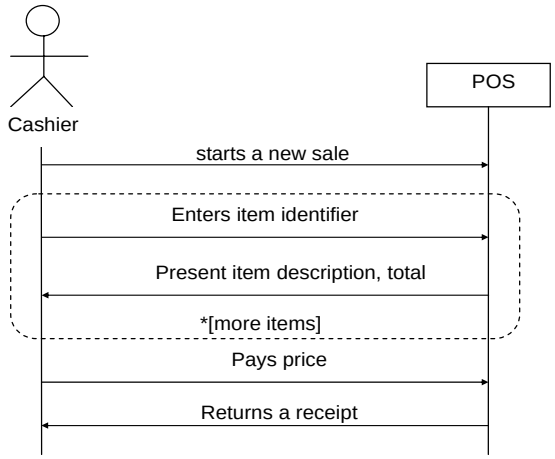

Figure 13. System sequence diagram for Process Sale (PS.SD)

Now that we have the system requirements and constraints specification in ALFReD notation as shown in Figure 14, we can perform the ReDLArch procedure. The following two sections show the first level design process and part of the second level design process.

POS
Observable Requirements POS.PS POS.HR
Non-observable Requirements None
Constraints C1. Different types of users need different user interfaces. C2. Maximum of 20 cashiers can use the system together

Figure 14. Logical component POS for input to ReDLarch

6.2 The First Level Design

Initially the whole system is the target of design.

Step 1) Perform buildability design

The first step is to utilize appropriate architectural patterns that show feasible solutions. By considering the constraint C1 that each type of actors has its own presentation, we decide to apply the MVC pattern that supports different views for different types of users. The structure obtained is shown in Figure 15.

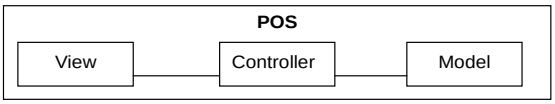

Figure 15. The structure after applying the MVC pattern

This step is followed by the step of fleshing out the introduced components. The result of applying PS use case scenario is shown in Figure 16.

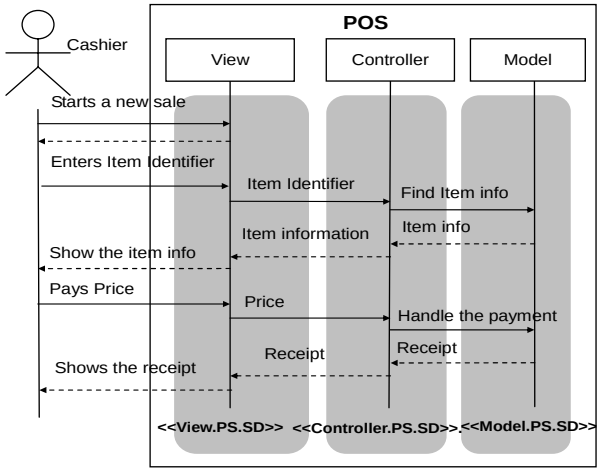

Figure 16. Fleshing out the structure of Figure 15

View	Controller	Model
View.PS.SD. View.HR.SD.	Controller.PS.SD. Controller.HR.SD.	Model.PS.SD. Model.HR.SD.
	C2.	

Figure 17. The result of buildability design

PS.SD is split into three sequence diagrams for View, Controller and Model components, respectively. Note that the constraint C2 was passed only to the Controller as shown in Figure 17. When we distribute requirements and constraints to components, we should consider whether they are completely disposed of by the structural design or they have been only partially taken care of by the structural design. In the latter case, they should be distributed to some or all the components for further handling.

Step 2) Perform reusability design

The first step for reusability design is to find out an appropriate existing component that can perform part of the functionality of a newly introduced component or to find the same functionality between two components to factor out common functionality.

In the current example, the developer finds out that there is an existing accounting component that handles payment and records sales. By this decision, the Model component is further partitioned into two components *Accounting Subsystem* for accounting and *Item Manager* for the rest of the functionality. The result is shown in Figure 18.

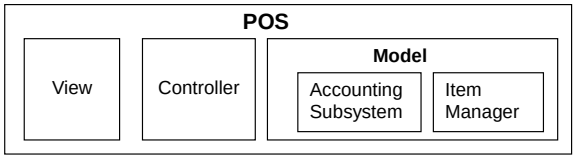

Figure 18. Components obtained after reusability design decision

Again this step is followed by the step of fleshing out the new components as in Figure 19 and Figure 20.

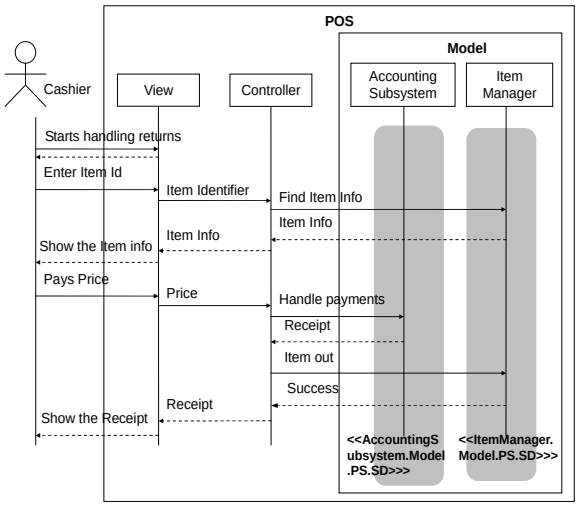

Figure 19. Fleshing out Figure 18

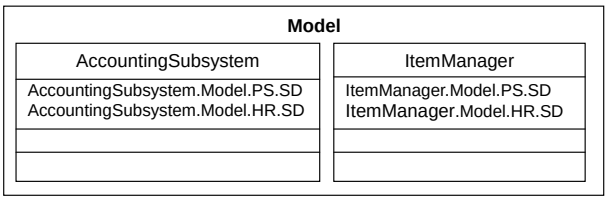

Figure 20. Result of applying reusability design to the Model component

Putting these results together, Figure 21 shows the logical architecture obtained from the first level design.

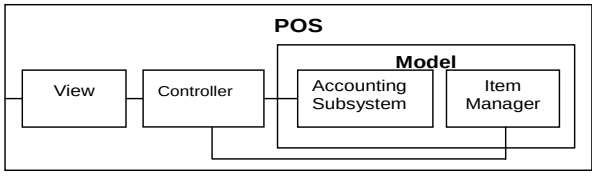

Figure 21.The logical architecture after the first iteration

6.3 The Second Level Design

While the other components are simple enough, the Controller component is complex and is selected for further design to handle the constraint C2. For buildability design, there are no suitable architectural patterns. So applying the second *if-then* part of the ReDLArch procedure, the controller component is decomposed into Request Controller and Service Controller components. By separating the business logic from the functionality of controlling customers' requests, it increases decoupling and make it possible to develop them independently. The result is shown in Figure 22. The step of fleshing out and the remaining steps are similar to the first-level design and are not shown here.

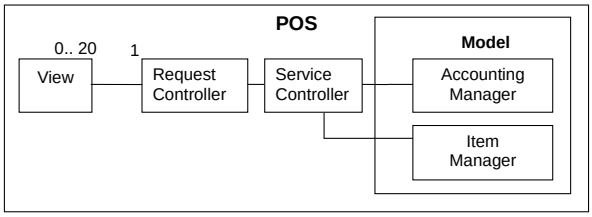

Figure 22. The logical architecture of the second iteration

7. Conclusion

In this paper, we presented an architecture description language for logical architecture ALFReD and its associated design method ReDLArch. Both ALFReD and ReDLArch are recursive, allowing top-down design. We applied them to a realistic system and showed their efficacy. The main characteristic of them are that they support recursive top-down design so that logical architectures for software systems of any scale and complexity can be designed.

There are several directions in which our work can be extended. To mention just a couple of them, firstly, we plan to build a tool that supports ALFReD and ReDLArch. Secondly, there are software architecture views other than logical view. Those views play their roles in appropriate phases of development. In addition to ALFReD and ReDLArch, the ADLs and the associated design methods for those views are needed to form a complete software architecture design framework.

8. References

[1] Bass, L., Clements, P., and Kazman, R., *Software Architecture in Practice*, 2nd ed., Addison-Wesley, 2003.

[2] Bass, L, Klein, M., and Bachman, F., "Quality Attribute Design Primitives and the Attribute Driven Design Method," Software Engineering Institute, Carnegie Mellon University, 2003.

[3] Booch, G., *Object-Oriented Analysis and Design with Applications*, The Benjamin/Cummings Publishing Co. Inc, 1994.

[4] Clements, P. C,, "A Survey of Architecture Description Languages," *Eighth Int'l Workshop on Software Specification and Design*, Germany, 1996.

[5] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., and Stafford, J., *Documenting Software Architectures: Views and Beyond*, Addison-Wesley, 2003.

[6] Garlan, D., and Allen, R., "Formalizing Architectural Connection," *Proc. 16th Int'l Conf on SE, Sorrento*, Italy, 1994.

[7] Hofmeister, C., Nord, R., and Soni, D., *Applied Software Architecture*, Addison-Wesley, 2000.

[8] Kruchten, P., "Architectural Blueprints - the '4+1' View Model of Software Architecture," *IEEE Software 12(6)*, Nov. 1995.

[9] Larman, G., *Applying UML and Pattern*, 2nd Ed., Prentice Hall, 2002.

[10] Luckham, D. C., et al, "Specification and Analysis of System Architecture using Rapide," *SIGSOFT Software Engineering Notes* 19(5), 1994.

[11] Medvidovic, N., et al., "Using Object-Oriented Typing to Support Architectural Design in the C2 Style," *Proc. ACM SIGSOFT '96 Fourth Symposium on the Foundations of Software Engineering*. p.24-32, ACM SIGSOFT. San Francisco, CA, Oct., 1996.

[12] Medvidovic, N., and Taylor, R. N., "A Classification and Comparison Framework for Software Architecture Description Languages," *IEEE Trans. Software Engineering, Vol. 26, No. 1*, Jan. 2000.

[13] Shaw, M., et al, "Abstractions for Software Architecture and Tools to Support Them," *IEEE Trans. Software Engineering 21(6)*, 1995.