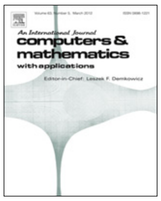
Computers and Mathematics with Applications 64 (2012) 1364–1375

Contents lists available at SciVerse ScienceDirect

Computers and Mathematics with Applications

journal homepage: www.elsevier.com/locate/camwa



A layer-based method for rapid software development

Lendy Lin[a], Weipang Yang[a], Jyhjong Lin[b,*]

[a] Department of Information Management, National Dong Hwa University, Hualien, Taiwan

[b] Department of Information Management, Ming Chuan University, Taoyuan, Taiwan

ARTICLE INFO

Keywords:
Software system
Development method
Layer-based
UML
Petri nets

ABSTRACT

A layer-based method for rapid software development is presented in this paper. It follows the guidelines suggested by Extreme Programming (XP) that require highly expressive programming languages (i.e., Java) and CASE tools. As in XP, this method addresses rapid software development for small- or medium-sized projects. Further, for effective guidance on the development, it directs the construction of system components by imposing an architecture-based concept of layered specification and construction of these components through its activities. Since the method follows the guidelines suggested by XP and supports effective guidance by a layered development of architectural components, team productivities can be greatly enhanced with less (but effective) overheads on specification work. The method uses UML and Petri nets as its modeling tool; for illustration, an example application is presented that specifies and directs the development of a software system with business-oriented Internet services.

1. Introduction

Properly identifying required activities and directing the completion of these activities for constructing relevant artifacts/deliverables are key issues for the successful development of a software system. To address these needs, it has become quite a matter of great importance for a software project team to impose a suitable development method because such a method can help to figure out required activities/artifacts, and more importantly, can provide information to assist on directing the completion/construction of these activities/artifacts. Although there are already plenty of software development methods in the literature and a common recognition that there are no methods that perfectly employ all sound features for effective development specification and guidance, it is still desired that suitable software development project methods can comprise the following features:

(1) It can define the structural and dynamic aspects of the development work where the structural aspect provides a means of specifying the required activities and relevant accessed artifacts/deliverables, and the dynamic aspect describes the behavioral effects of these activities on each other and relevant artifacts/ deliverables.
(2) It features proper mechanisms to support effective guidance on the development work, e.g., directing such effects as execution sequences of activities and construction relevance of artifacts/deliverables.
(3) The defined activities can be considered in a leveled manner such that team members can participate in the execution by providing/monitoring respective information about the specification and completion of activities/artifacts.
(4) It can support rapid development for small- or medium-sized projects by using less overhead effectively for completing/constructing defined activities/artifacts; automatic tools should also be generated to facilitate its practical applications.

* Corresponding author.
E-mail addresses: lendy.lin@gmail.com (L. Lin), wpyang@mail.ndhu.edu.tw (W. Yang), jlin@mail.mcu.edu.tw (J. Lin).

0898-1221/$ – see front matter © 2012 Elsevier Ltd. All rights reserved.
doi:10.1016/j.camwa.2012.03.082

As mentioned earlier, there are already plenty of software development methods in the literature. Among them, for instance, Waterfall, Evolutionary Development, and Component-based Software Engineering are three traditional generic models [1–8] that are not mutually exclusive but often used together for development of large systems; Rational Unified Process (RUP) [9,10] and Spiral model [11,12] give an illustration that employs elements of these three models. Some formal approaches such as *B* and Cleanroom models can also be found in [13–16] that emphasize the mathematical specification for the software system and its mapping directly into an implementation. With respect to these design-oriented (featuring more analysis and design work) models, many alternatives that criticize the overhead on design have been proposed (e.g., including less or even no analysis and design work) such as Agile Development [17–19], Rapid Application Development (RAD) [20] and Extreme Programming (XP) [21,22], where code work (i.e., implementation, testing, and refactoring) are the focus. Whereas such design- or code-oriented models try to optimize extremes in very different spectrums, some compromise models can be found in [23] that employ simplified design and code work through rapid engineering means, where SMALLTALK and two CASE tools are imposed on the development process.

In general, these existing approaches provide sound mechanisms for development specification and guidance; nonetheless, some drawbacks with respect to the aforementioned desired features can still be found among them:

1. Both the design- and code-oriented models go to the extremes in their relevant spectrums, so the advantages for one would become the opposite for the other [23,24]. For instance, design-oriented models take advantage of analysis and design work but result in more design overhead that impedes the rapid development for small- or medium-sized projects; in contrast, code-oriented models focus on code work for rapid development but offer less support on information about defined activities/artifacts, which is essential for team members to participate in the execution of the development process.
2. For the comprise approach that takes advantage of both extremes, its lower overhead on design work and associated rapid engineering means make it good for the rapid development for small- or medium-sized projects. However, the limited design work on the other hand results in lack of sufficient mechanisms for supporting effective guidance on the development work (e.g., directing the executions of activities in accordance with affecting artifacts/deliverables toward realizing user requirements) [21,25,26].
3. The existing approaches essentially focus on the specification and completion of defined activities/artifacts; in contrast, few considerations about the management of these tasks can be found in their statements (e.g., providing information or monitoring the status of the specification/completion of these activities/artifacts) [21,24,26,27]. To our knowledge, however, such management issues are not negligible, since they play a critical role in the success of these tasks.

To address these deficiencies, we present in this paper a layer-based method that supports rapid software development for small- or medium-sized projects. In general, this method follows the guidelines suggested by XP that require highly expressive programming languages (i.e., Java) and CASE tools. As in XP, this method requires less design work, so that software development is speedy. However, for providing guidance on development, it focuses on the construction of system components by imposing an architecture-based concept of layered specification and construction of these components on its activities. This makes it easy to direct the development work by taking into account the effects of executions of these activities on these components for realizing desired user requirements. Since the method follows the guidelines in XP and supports effective guidance by a layered development of architectural components, team productivities can be greatly enhanced by less (but effective) overhead on specification work. Finally, the concept of developing architectural components in a layered manner also assists in the management issues, since various activities can be considered in a corresponding layered manner such that responsible team members can participate in the project by providing information or monitoring status about the specification and completion of these activities (e.g., determining and analyzing the status of these activities). For practical applications, the method uses UML [28,29] and extended Petri nets (PN) [27,30,31] as its modeling mechanisms, and, based on the semantics in PN, a software development CASE tool is constructed to support the layered creation of design diagrams and their mapping into executable Java code.

This paper is organized as follows. Section 2 presents the method and its corresponding models in UML and PN. For illustration, an example application is presented in Section 3 that specifies and directs the development of a software system with business-oriented Internet services. Finally, Section 4 presents conclusions.

2. The development method

As shown in Fig. 1, the method is layer-based with the following four steps:

1. *use case identification*, described in a UML use case diagram, that clarifies user requirements to be satisfied in the system;
2. *architectural specification*, described in three design diagrams (a PN process, a UML component, and a PN behavior), that determines a layered identification process for clarifying and specifying architectural components of the system that collaboratively support the realization of each desired use case;
3. *component construction*, in Java code, that implements and tests these architectural components for realizing the desired use case;
4. *iterations for specification and construction of layered components*, achieved by a sequence of creating design diagrams and linking them with Java code in accordance with the order prescribed in the identification process, that realizes the

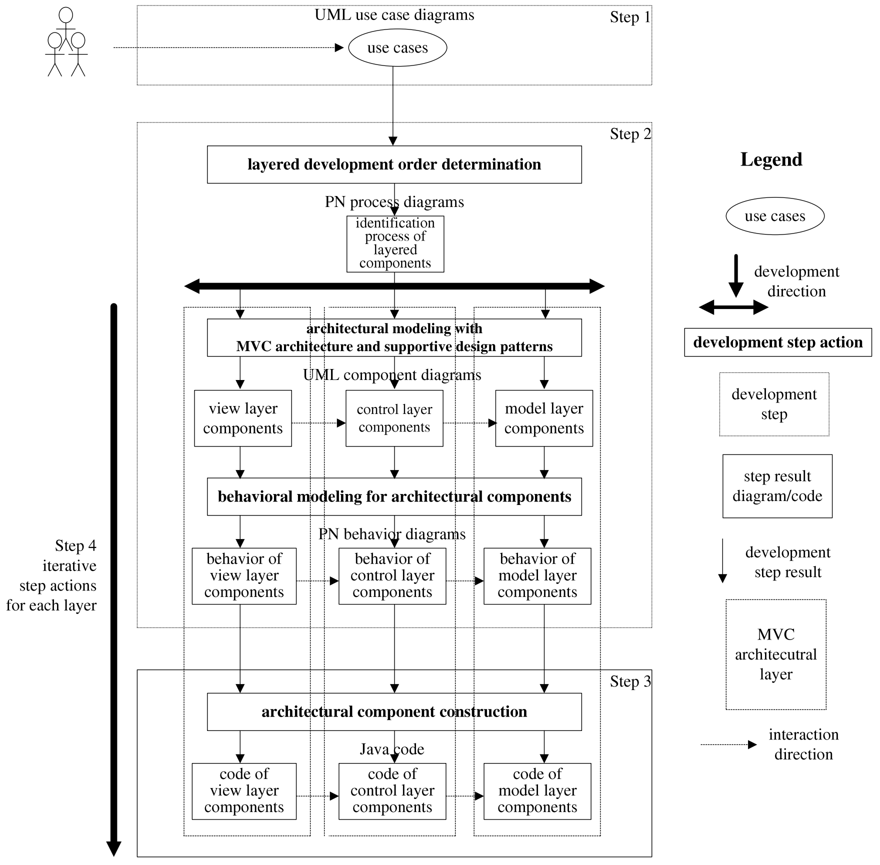

UML use case diagrams
Step 1
use cases
Step 2
layered development order determination
PN process diagrams
identification process of layered components
architectural modeling with MVC architecture and supportive design patterns
UML component diagrams
view layer components
control layer components
model layer components
behavioral modeling for architectural components
PN behavior diagrams
behavior of view layer components
behavior of control layer components
behavior of model layer components
Step 4 iterative step actions for each layer
Step 3
architectural component construction
Java code
code of view layer components
code of control layer components
code of model layer components
Legend
use cases
development direction
development step action
development step
step result diagram/code
development step result
MVC architecutral layer
interaction direction

Fig. 1. The layer-based development method.

desired use case by specifying and constructing architectural components through steps 2 and 3 iteratively in a layered manner;

where steps 2 and 3 are necessarily repeated as an iterative procedure (described in step 4) for each identified use case; in addition, for the four diagrams employed: the first one identifies all desired use cases and the remaining three are used at step 2 iteratively to complete the essential design work for each identified use case.

2.1. *The use case identification*

The first step is to identify user requirements that need to be satisfied in the system. As in many existing approaches, all desired user requirements can be identified by various techniques such as interviews, questionnaire, observation, brainstorming meeting, etc. and the identified requirements are represented by means of use cases in a UML use case diagram. Note that the reader is referred to [28] for more detail about use cases and their representation in UML.

After identifying desired use cases, it is good time to perform design work for constructing them in an incremental manner; that is, based on available resources and under possible non/functional considerations, these use cases can be constructed according to an incremental plan where they are prescribed into a set of increments for design and construction for a controlled series of deliverable releases. The following describes the two steps for design and construction of each use case or increment.

2.2. *The architectural specification*

For each desired use case/increment, its design work begins by identifying an architectural partitioning of physical components that collaboratively support the realization of this use case/increment. To our best knowledge, clarifying system components and their interactions is a most suitable means to ensure the achievement of desired use cases [32,33]. In this step, we take advantage of frameworks at the architectural design level [34] that support rapid development throughout the following activities. Therefore, based on system characteristics and/or desired functional/non-functional requirements, these components may be identified by applying some well-known architectural frameworks such as MVC [35–38], MFC [39,40], and PCMEF [41]. Among these frameworks, MVC (Model, View, and Control) is most commonly chosen for development of small- or medium- sized projects due to its well-recognized role-based concept, a classic use of separation of concerns in object-oriented design, and the most supportive design patterns [42]. Thus, MVC is applied herein in a layered manner such that components are allocated respectively at its three layers and identified layer by layer (e.g., from View to Control to Model) according to some designated order. Note that supportive design patterns may be used at each layer to enhance the effectiveness of these components toward the realization of this use case/increment.

When architectural components are identified in a layered manner, three design diagrams are used to specify the identification process as well as those components identified at each working layer and their interactions to achieve this use case/ increment: (1) a PN process diagram that describes the identification process and any components to be identified at each layer during the process; (2) a UML component diagram that denotes the identification process by detailed specification of those components identified at each layer and their dependent relationships; and (3) a PN behavior diagram that denotes in turn the component diagram by description of how those components identified at each layer interact with each other to achieve this use case/increment. It is noticed that for any layer whose components have not yet been identified, a substitute stub is imposed on that layer for completing the structural and behavioral specification throughout the layered identification process.

2.3. The component construction

While architectural components for a use case/increment are identified and their structures/behaviors are specified in a layered manner, their construction can be achieved directly by linking the design work (i.e., PN behavior diagrams) into executable Java code for supporting rapid development. In general, the linkage can be achieved by examining the formal semantics in PN behavior diagrams [27,30,31] to transform behavioral interactions among various components into respective behavioral skeletons in Java code for realizing the use case/increment where further detailed execution statements can then be realized in each skeleton. Note that while system components are constructed by suitable transformations and detailed implementations, their quality in realizing the use case/increment is critical and hence needs to be ensured by corresponding tests; therefore, testing for individual components, interactions among various components, and use case-compliance throughout all components needs to be imposed along with this construction work, for which many well-known object-oriented testing techniques [43–47] may be applied.

2.4. Iterations for specification and construction of layered components

In the method, steps 2.2 and 2.3 are repeated as an iterative procedure for completing specification and construction of those components at the three MVC layers for realizing a desired use case/increment. That is, components and their interactions in the three MVC layers are identified, specified, and constructed iteratively according to some designated order. Note that for interaction–or process–or data-oriented use cases/increments, various orders may be applied due to their different focuses; for instance, for a use case/increment that emphasizes the interactions between the software system and users, interaction and data models play critical roles for its realization, and hence architectural components can be identified by the order View $\rightarrow$ Model $\rightarrow$ Control or Model $\rightarrow$ Control $\rightarrow$ View to address such characteristics. Also, since the development work follows a layered path, it is necessary to impose a substitute stub on any layer whose components have not yet been identified for completing the specification of all components to achieve desired behaviors. Since architectural components are specified in a layered manner and then constructed directly by linking their specification into executable Java code, a rapid development process can be effectively achieved.

It is noticed that since system components are identified and developed in a layered manner, two more advantages can be found from such a process: (1) the layer-based design and layered development make it easy to support effective guidance on the development work (e.g., the design/code activities can be directed in a designated order of their effects on system components toward the realization of desired use cases/increments); and (2) the layered development helps also the management of the development work by facilitating the participation of team members in a corresponding layered manner where these members may provide information about the executions of the design/code activities or their accessed components (e.g., the execution status of these activities or the versions of their accessed components) and then monitor them by analyzing the PN process diagram based on its formal semantics (e.g., with its PN-based formal semantics, analysis can be done via decision procedures that traverse its derived reachability graph [48]).

3. An illustration

In this section, we illustrate the method by specifying and directing the development of a software system with services for book publishing.

3.1. The use case identification for book publishing

The first step is to identify user requirements (use cases) for providing customers with services related to book publishing. Many existing book publishing companies (via their own websites or some intermediary agents, e.g., Amazon) offer not only

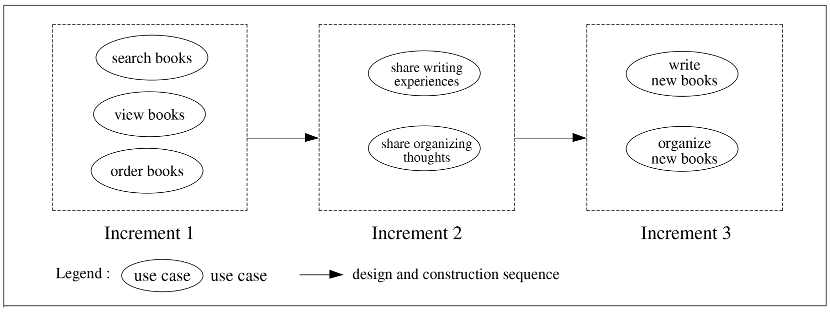


Fig. 2. Use cases and increments for book publishing.

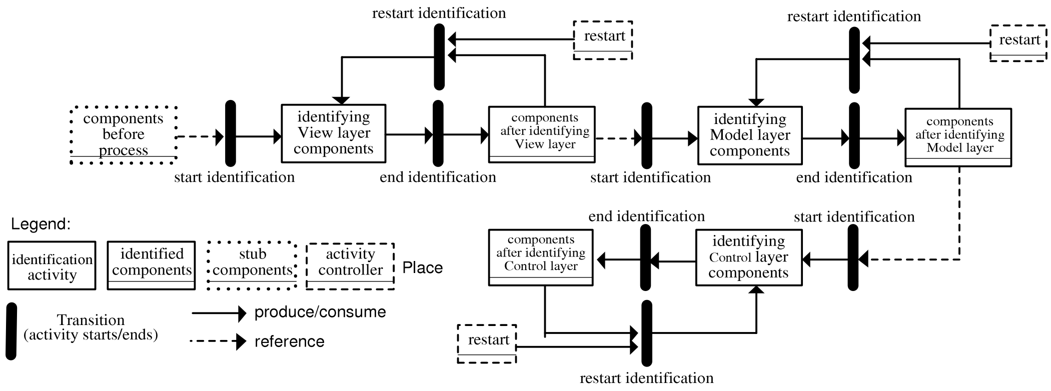


Fig. 3. Identification process of layered components for use case—share writing experiences.

ordinary services such as searching, viewing, and ordering books, but also sophisticated ones such as sharing of writing-experiences/organizing-thoughts and writing/organizing of new books. Therefore, in our example, seven use cases can be identified and, based on their relevance, they can be prescribed into three increments for further design and construction (with a designated sequence of deliverable releases). Fig. 2 is an UML use case diagram with extensions to describe these seven use cases, prescribed increments, and designated design/construction sequences.

3.2. The architectural specification for book publishing

For each desired use case/increment, its design work begins by identifying an architectural partitioning of physical components that collaboratively support its realization. In this step, these architectural components can be identified by the following sub-steps:

1. Determining the identification order for any components among those layers in the architectural framework employed (e.g., in our selected MVC framework, the identification can have such orders as View→ Control→ Model or vice versa). In general, this can be achieved by referencing the characteristics of the use case/increment where each MVC layer addresses certain aspects (e.g., for interaction- or process- or data-oriented use cases/increments, various identification orders may be applied due to their different focuses). For instance in our example, since Increment 2 emphasizes the interactions among customers for sharing experiences/thoughts where interaction and data models play critical roles for its realization, architectural components can therefore be identified by the order View→ Model → Control to address such characteristics.
2. Under the prescribed order, work on each layer in sequence to identify its comprising architectural components and their dependent relationships (among themselves or with other components in next layers). In our example, as shown in the PN process diagram in Fig. 3, layered components for the 'share writing experiences' use case in Increment 2 are identified under the View→ Model→ Control order where Figs. 4a and 4b illustrate detailed specification of these components identified at each layer, as shown in Fig. 3. It is also noted that, as shown in Fig. 4b, supportive DAO (Data Access Object) and CMD (Command) design patterns [36] are used in Model and Control layers respectively to enhance the effectiveness of these components for the realization of this use case.
3. While a UML component diagram is used to specify the structure of architectural components in a layer, a PN behavior diagram that in turn denotes the component diagram is then imposed to describe how these components interact among each other (or with other components in next layers) to achieve the role that this layer plays. For our example, Figs. 4a and 4b are the two component diagrams for detailed specification of those components identified through the prescribed

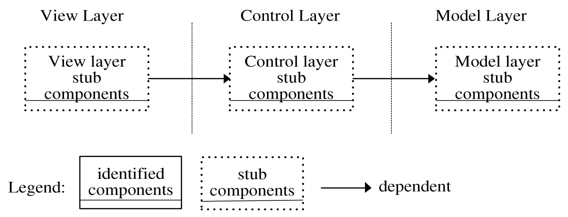


Fig. 4a. Detailed specification for 'components before process' in Fig. 3.

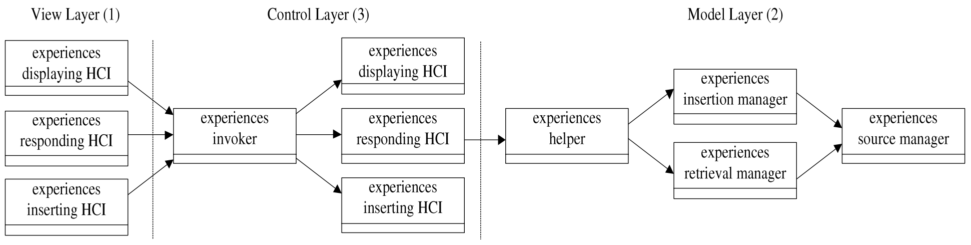

View Layer (1)
Control Layer (3)
Model Layer (2)
experiences displaying HCI
experiences responding HCI
experiences inserting HCI
experiences invoker
experiences displaying HCI
experiences responding HCI
experiences inserting HCI
experiences helper
experiences insertion manager
experiences retrieval manager
experiences source manager

Fig. 4b. Specification for 'components after identifying View/Model/Control layer' in Fig. 3.

View → Model→ Control order for the 'share writing experiences' use case, while Fig. 5 is the behavior diagram that denotes in turn Fig. 4b to describe through the same order how these components interact to achieve the role that each layer plays for realizing this use case.

3.3. *Component construction for book publishing*

While components for the 'share writing experiences' use case are identified and their structural/behavioral descriptions are specified in a prescribed order, their construction can be achieved directly by transforming these specifications (i.e., PN behavior diagrams) into executable Java code for supporting rapid development. Particularly, the transformation is achieved by examining the formal semantics in PN behavior diagrams to transform behavioral interactions among various components into respective behavioral skeletons in Java code where further detailed execution statements can then be constructed in each skeleton.

3.4. *Iterations for specification and construction of layered components for book publishing*

In this method, steps 3.2 and 3.3 are repeated as an iterative procedure for completing specification and construction of those components at the three MVC layers for realizing the 'share writing experiences' use case. That is, components and their interactions at the Model, View, and Control layers are identified, specified, and constructed according to a prescribed View → Model → Control order. Also, since the development work follows a layered manner, substitute stubs are imposed on any layers whose components have not yet been identified for completing the specification of all components along the layered identification process.

Further, as stated in step 2.4 and illustrated in Fig. 3, the layer-based process supports the specification and construction of system components in accordance with the layered executions of its activities, therefore it also supports effective guidance on the management of these activities by providing team members with information about these activities and/or accessed components (e.g., their execution status and/or versions of accessed components) in a corresponding layered manner and then monitoring them by analyzing the PN process diagram based on its formal semantics. For instance, with its PN-based formal semantics in Fig. 3, analysis (e.g., keeping track of the execution status of each activity) can be done via decision procedures that traverse its derived reachability graph as shown in Fig. 6 where states 2, 4, and 6 represent, respectively, one working status of specifying and constructing components at a layer; these states may be reentered in case iterations are needed for accommodating changes around the design or code work for refactoring design diagrams or executable Java code.

3.5. *Tool supports*

With less overhead on specifying and constructing system components and effective guidance on the management of these activities, this method can support rapid development for small-or medium-sized projects. However, automatic tools should also be generated to facilitate its practical applications. As such, we illustrate herein a software development CASE tool accompanied by this method as follows: (1) Fig. 7 is the PN process diagram created by the CASE tool that describes the identification process as shown in Fig. 3; since it employs the formal semantics of PN, analysis of the execution status of the

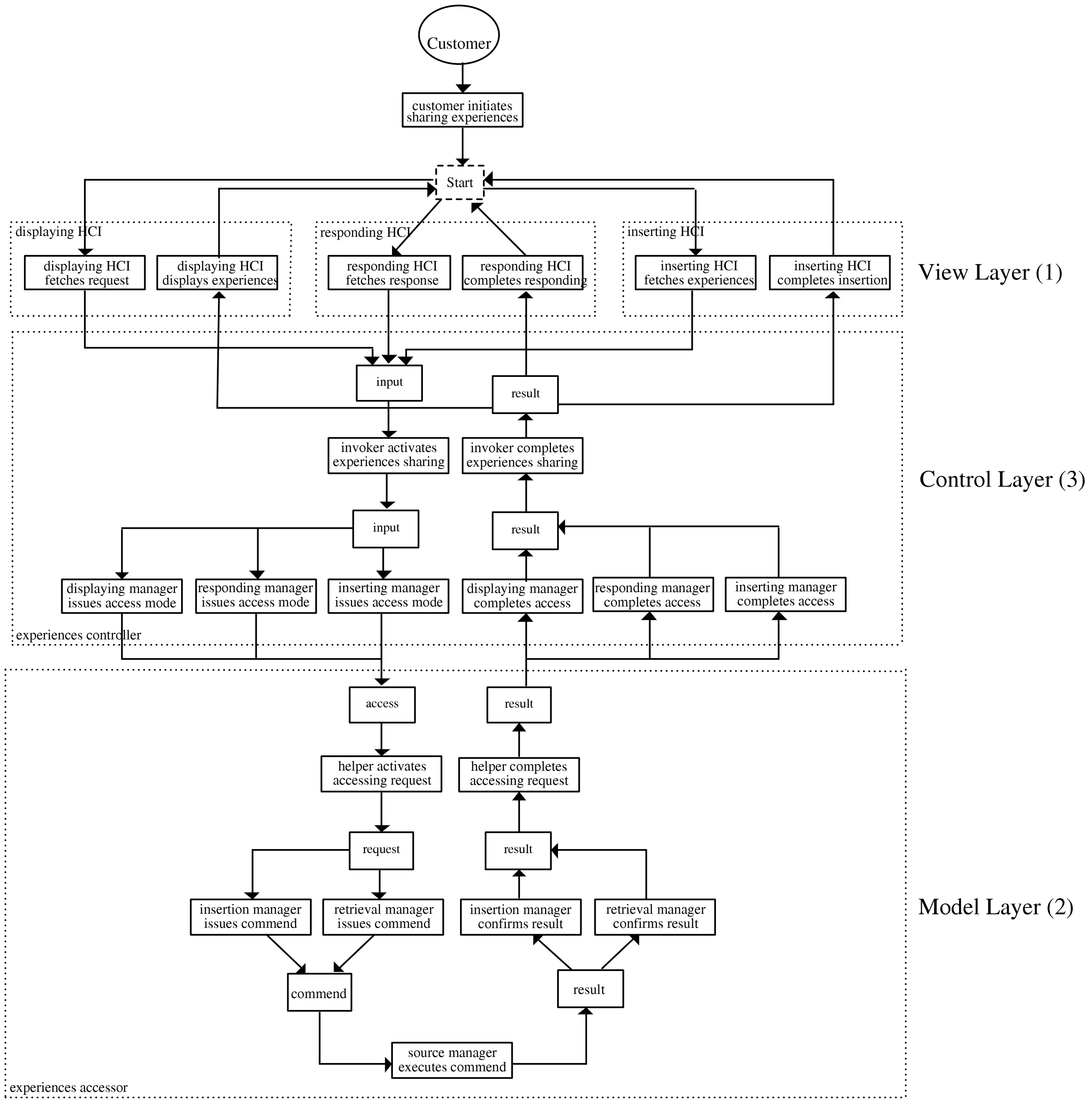

Customer
customer initiates sharing experiences
Start
displaying HCI
displaying HCI fetches request
displaying HCI displays experiences
responding HCI
responding HCI fetches response
responding HCI completes responding
inserting HCI
inserting HCI fetches experiences
inserting HCI completes insertion
View Layer (1)
input
result
invoker activates experiences sharing
invoker completes experiences sharing
Control Layer (3)
input
result
displaying manager issues access mode
responding manager issues access mode
inserting manager issues access mode
displaying manager completes access
responding manager completes access
inserting manager completes access
experiences controller
access
result
helper activates accessing request
helper completes accessing request
request
result
insertion manager issues commend
retrieval manager issues commend
insertion manager confirms result
retrieval manager confirms result
Model Layer (2)
commend
result
source manager executes commend
experiences accessor

Fig. 5. Behavioral specification for components in Fig. 4b.

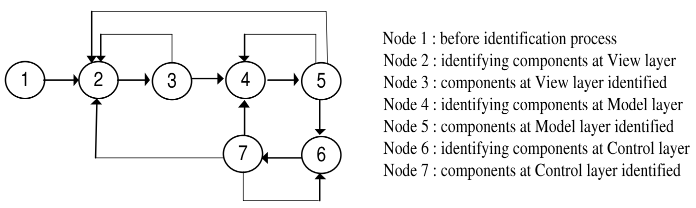
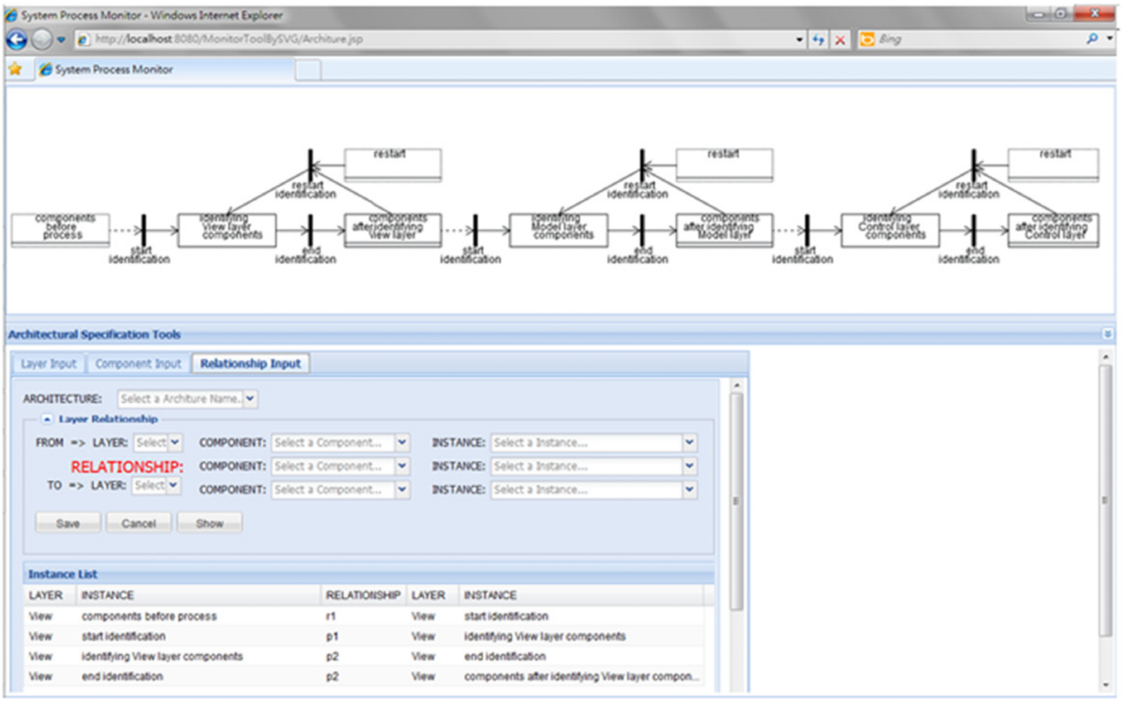

1
2
3
4
5
6
7
Node 1 : before identification process
Node 2 : identifying components at View layer
Node 3 : components at View layer identified
Node 4 : identifying components at Model layer
Node 5 : components at Model layer identified
Node 6 : identifying components at Control layer
Node 7 : components at Control layer identified

Fig. 6. State reachability graph derived from the identification process in Fig. 3.

process can be achieved by traversing its derived reachability graph as shown in Fig. 6; (2) Based on Fig. 7, that describes a sequence of identification activities, Fig. 8 illustrates a tool-created component diagram that denotes Fig. 7 by detailed specification of the 'components after identifying View/Model/Control layer' in Fig. 7; (3) Based on Fig. 8, that describes


System Process Monitor - Windows Internet Explorer
http://localhost:8080/MonitorToolBySVG/Archture.jsp
Bing
System Process Monitor
restart
restart identification
components before process
start identification
identifying View layer components
end identification
components after identifying View layer
identifying Model layer components
components after identifying Model layer
identifying Control layer components
components after identifying Control layer
Architectural Specification Tools
Layer Input
Component Input
Relationship Input
Save
Cancel
Show

ARCHITECTURE: Select a Archture Name...

Layer Relationship

FROM => LAYER: Select COMPONENT: Select a Component... INSTANCE: Select a Instance...

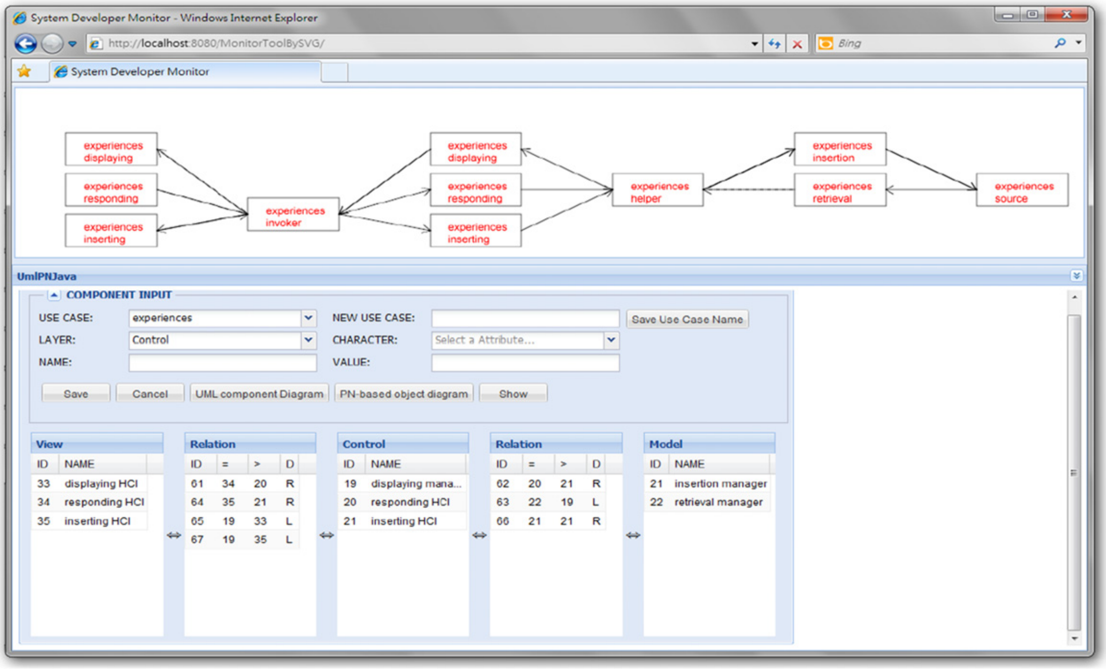
RELATIONSHIP: COMPONENT: Select a Component... INSTANCE: Select a Instance...

TO => LAYER: Select COMPONENT: Select a Component... INSTANCE: Select a Instance...

Instance List

LAYER	INSTANCE	RELATIONSHIP	LAYER	INSTANCE
View	components before process	r1	View	start identification
View	start identification	p1	View	identifying View layer components
View	identifying View layer components	p2	View	end identification
View	end identification	p2	View	components after identifying View layer compon...

Fig. 7. Tool-created identification process for use case—share writing experiences as in Fig. 3.



COMPONENT INPUT

USE CASE: experiences NEW USE CASE:

LAYER: Control CHARACTER: Select a Attribute...

NAME: VALUE:

View

ID	NAME
33	displaying HCI
34	responding HCI
35	inserting HCI

Relation

ID	=	>	D
61	34	20	R
64	35	21	R
65	19	33	L
67	19	35	L

Control

ID	NAME
19	displaying mana...
20	responding HCI
21	inserting HCI

Relation

ID	=	>	D
62	20	21	R
63	22	19	L
66	21	21	R

Model

ID	NAME
21	insertion manager
22	retrieval manager

Fig. 8. Tool-created detailed specification as in Fig. 4b.

the structural part of system components, Fig. 9 illustrates a tool-created behavior diagram that denotes, in turn, Fig. 8 to describe how these components interact to achieve the realization of the 'share writing experiences' use case; and (4) Based on Fig. 9, that describes the behaviors of system components, Fig. 10 illustrates a tool-created Java code that results from the transformation of the behaviors in Fig. 9. Note that the transformation is achieved by checking the PN-based semantics in Fig. 9 to transform behavioral interactions among various components into respective behavioral skeletons in Java code where further detailed execution statements can then be realized in each skeleton.

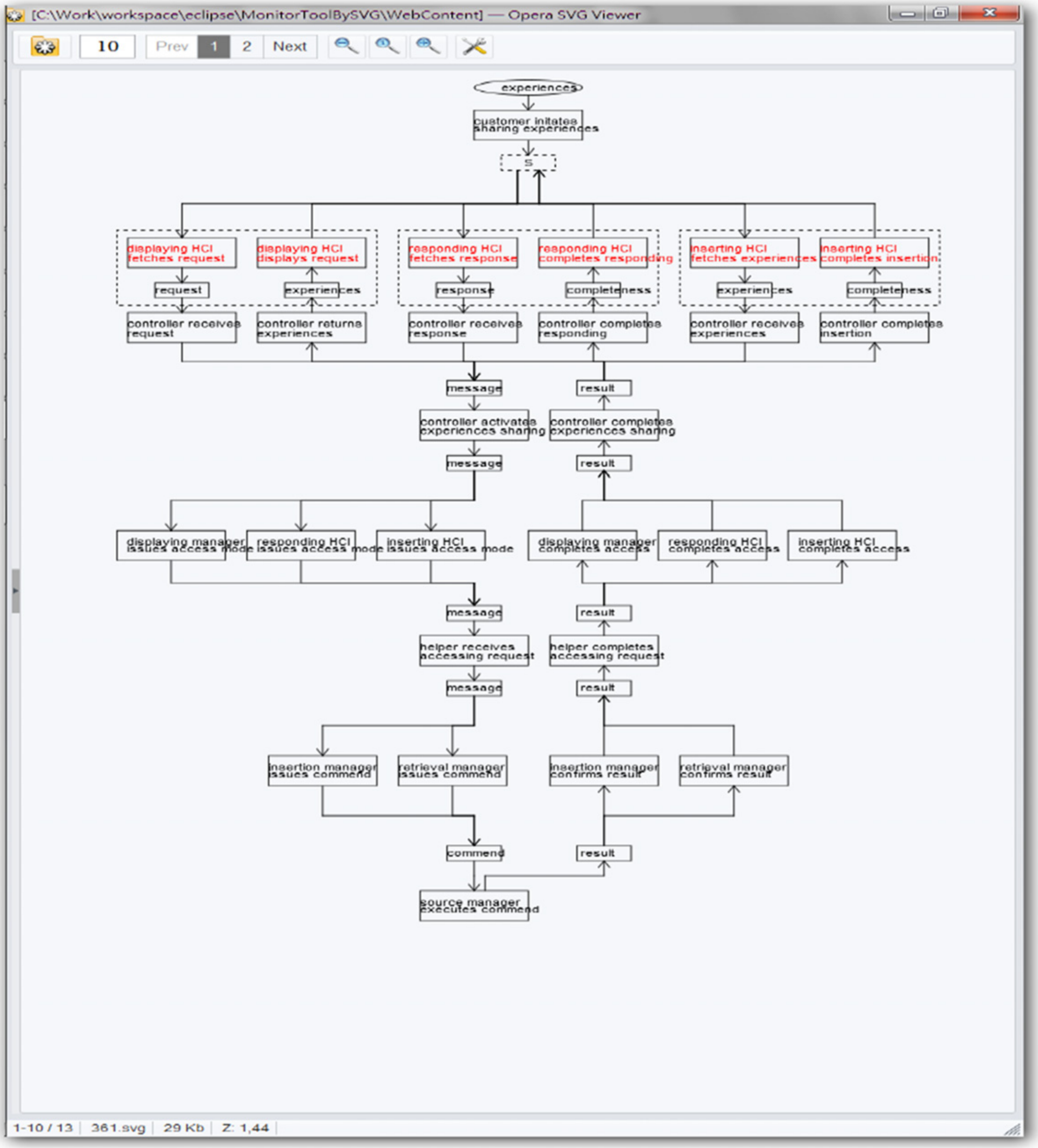
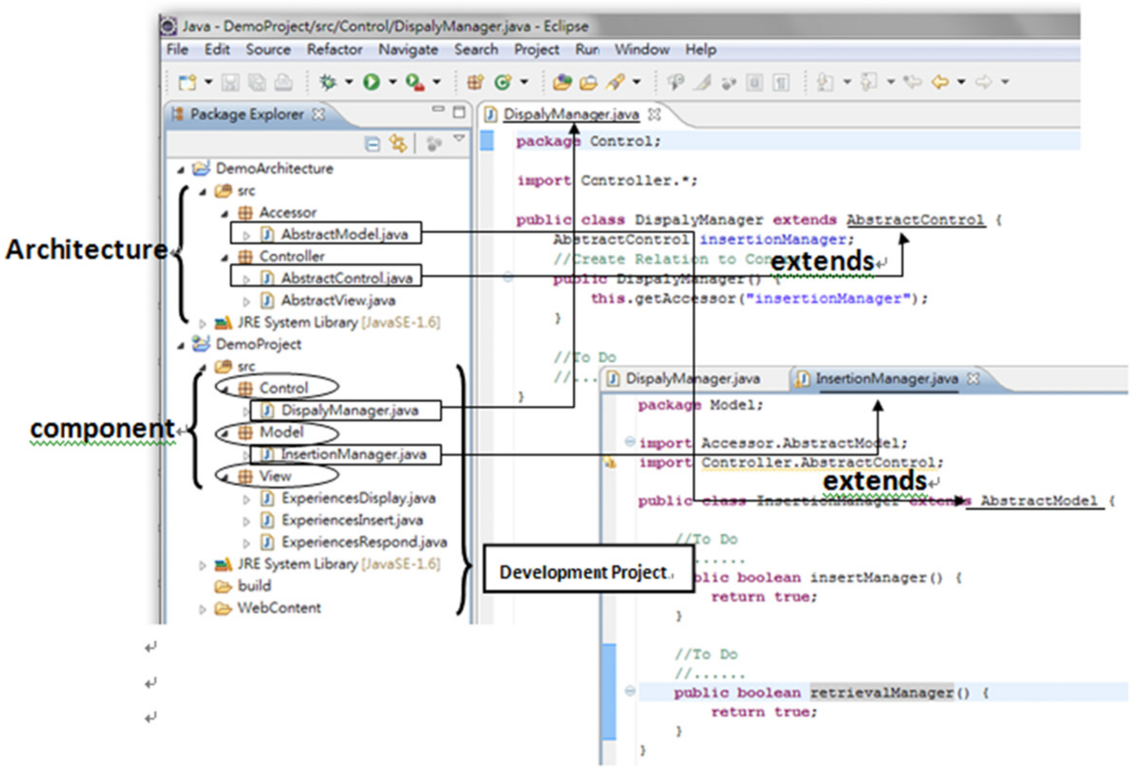

[C:\Work\workspace\eclipse\MonitorToolBySVG\WebContent] — Opera SVG Viewer
10
Prev
1
2
Next
experiences
customer initiates sharing experiences
S
displaying HCI fetches request
displaying HCI displays request
responding HCI fetches response
responding HCI completes responding
inserting HCI fetches experiences
inserting HCI completes insertion
request
experiences
response
completeness
experiences
completeness
controller receives request
controller returns experiences
controller receives response
controller completes responding
controller receives experiences
controller completes insertion
message
result
controller activates experiences sharing
controller completes experiences sharing
message
result
displaying manager issues access mode
responding HCI issues access mode
inserting HCI issues access mode
displaying manager completes access
responding HCI completes access
inserting HCI completes access
message
result
helper receives accessing request
helper completes accessing request
message
result
insertion manager issues commend
retrieval manager issues commend
insertion manager confirms result
retrieval manager confirms result
commend
result
source manager executes commend
1-10 / 13
361.svg
29 Kb
Z: 1,44

Fig. 9. Tool-created behavioral specification as in Fig. 5.

3.6. Summary

Based on the above illustration, we may summarize below how this method addresses those aforementioned deficiencies in existing approaches.

1. This method is a compromise approach that takes advantage of the two extreme existing approaches of design- or code-oriented models. In general, it demands less analysis and design work so as to speeding software development; further, it imposes an architecture-based concept of layered specification and construction of system components on its activities to support more information about defined activities/artifacts for facilitating team members participation in the execution of the development process.
2. With the specification and construction of system components in a layered manner, this method provides effective guidance for the development work by directing the executions of its activities in accordance with their effects on system components for realizing desired user requirements. Since the method demands less design work and supports effective



```
package Control;

import Controller.*;

public class DispalyManager extends AbstractControl {
    AbstractControl insertionManager;
    //Create Relation to Con
    public DispalyManager() {
        this.getAccessor("insertionManager");
    }

    //To Do
    //...
}
```

```
package Model;

import Accessor.AbstractModel;
import Controller.AbstractControl;

public class InsertionManager extends AbstractModel {

    //To Do
    //......
    public boolean insertManager() {
        return true;
    }

    //To Do
    //......
    public boolean retrievalManager() {
        return true;
    }
}
```

Fig. 10. Part of Java code for use case—share writing experiences.

guidance by a layered development of architectural components, team productivities can be greatly enhanced through lower, but effective, investment of overhead to specification work.

3. Finally, with the layered specification and construction of system components, this method also helps in the management of the development work by facilitating the participation of team members in a corresponding layered manner where they may provide information about the executions of their development activities or relevant accessed components (e.g., the execution status of these activities or the versions of relevant accessed components) and then monitor these activities by analyzing the PN process diagram based on its formal semantics. That is, with its PN-based formal semantics, analysis can be done via decision procedures that traverse its derived reachability graph.

4. Conclusions

In this paper, we present a layer-based method that supports rapid software development for small- or medium-sized projects. This method follows the guidelines suggested by XP with a software development CASE tool for modeling design diagrams and mapping into executable Java code. As in XP, it demands less design work to speed software development. Further, for providing guidance on development, it employs a layered specification and construction of system components based on a selected MVC architectural framework; this makes it easy to direct the development work by considering the executions of process activities regarding their effects on system components for realizing desired user requirements.

For illustration, the method is applied to specify and direct the development of a software system with services for book publishing. More specifically, a 'share writing experiences' use case is illustrated by identifying its architectural components through the View→Model→Control order based on its interaction-oriented characteristics. In particular, while architectural components are identified in a layered manner, supportive design patterns like DAO and CMD are employed at respective layers to enhance the effectiveness of these identified components on the realization of this use case. Finally, when system components are identified and specified in structural and behavioral diagrams, their construction is achieved by the CASE tool that maps these visual diagrams into executable Java skeleton for further detailed implementation.

Currently, we are now advancing the software development CASE tool so that it can support a round-trip engineering process; that is, the tool is eventually expected to support dynamic refactoring around modeling and coding levels where changes at the modeling level can result in corresponding updates at the coding level and vice versa. This will enhance the effectiveness of our proposed method by taking full advantages of its rapid development characteristics based on a selected architectural framework and the corresponding sequence for a layered identification and specification of architectural components. Afterwards, we will study also the application of this enhanced method and tool on the development of various kinds of software systems with interaction-/process-/data-oriented characteristics, such as social network systems, business transaction systems, decision support systems, and executive information systems. While presenting different characteristics, the development of these systems would result in different identification and specification sequences for their architectural components. While developing these systems, experiences from the application of this method can be correspondingly collected for validating its usefulness and effectiveness.

Finally, since this method supports the management of the development work by facilitating team members to provide information about the executions of the design/code works or their accessed components and then to monitor them by analyzing the PN process diagram based on its formal semantics, we will therefore further explore its effects on the management of the development work where the provision and/or monitoring of information about the executions of the design/code works or their accessed components may be realized by certain useful Java documentation tools such as Annotations and Javadoc.

References

[1] Walker Royce, Managing the development of large software systems: concepts and techniques, in: Proc. of IEEE Westcon, August 1970, pp. 1–9.

[2] Dorothy Graham, Incremental development and delivery for large software systems, in: Proc. of IEE Colloquium on Software Prototyping and Evolutionary Development, November 1992, pp. 2/1–2/9.

[3] John Crinnion, The evolutionary development of business systems, in: Proc. of IEE Colloquium on Software Prototyping and Evolutionary Development, November 1992, pp. 3/1–3/11.

[4] Jim Ning, Component-based software engineering, in: Proc. of 5th IEEE International Symposium on Assessment of Software Tools and Technologies, June 1997, pp. 34–43.

[5] James Highsmith, Adaptive Software Development: A Collaborative Approach to Managing Complex Systems, Dorset House, New York, 2000.

[6] Xia Cai, Michael Lyu, Kam-Fai Wong, Roy Ko, Component-based software engineering: technologies, development frameworks, and quality assurance schemes, in: Proc. of IEEE APSEC, December 2000, pp. 372–379.

[7] Anneke Kleppe, Jos Warmer, Wim Bast, MDA Explained: The Model Driven Architecture: Practice and Promise, Addison Wesley Professional, Indianapolis, Indiana, 2003.

[8] Xinyu Zhang, Li Zheng, Cheng Sun, The research of the component-based software engineering, in: Proc. of 6th IEEE International Conference on Information Technology, April 2009, pp. 1590–1591.

[9] Philippe Kruchten, The Rational Unified Process, Addison Wesley Professional, Indianapolis, Indiana, 1999.

[10] Philippe Kruchten, The Rational Unified Process: An Introduction, Addison Wesley Professional, Indianapolis, Indiana, 2000.

[11] Barry Boehm, A spiral model of software development and enhancement, IEEE Computer 21 (5) (1988) 61–72.

[12] Chonchanok Viravna, Lessons learned from applying the spiral model in the software requirements analysis phase, in: Proc. of 3th IEEE International Symposium on Requirements Engineering, January 1997, p. 40.

[13] Harlan Mills, Michael Dyer, Richard Linger, Cleanroom software engineering, IEEE Software 4 (5) (1987) 19–25.

[14] Richard Linger, Cleanroom process model, IEEE Software 11 (2) (1994) 50–58.

[15] John Wordsworth, Software Engineering with *B*, Addison Wesley, Harlow, England, 1996.

[16] Stacy Prowell, Carmen Trammell, Richard Linger, Jesse Poore, Cleanroom Software Engineering: Technology and Process, Addison Wesley Professional, Indianapolis, Indiana, 1999.

[17] Pekka Abrahamsson, Outi Salo, Jussi Ronkainen, Juhani Warsta, Agile Software Development Methods, VTT Publications, Espoo, Finland, 2002.

[18] Agile. http://www.agilealliance.org/ (accessed on: January 2011).

[19] Robert Martin, Agile Software Development: Principles, Patterns, and Practices, Prentice Hall, New Jersey, 2003.

[20] Jennifer Stapleton, Dynamic Systems Development Method, Pearson Education, New Jersey, 1997.

[21] Lowell Lindstrom, Ron Jeffries, Extreme programming and agile software development methodologies, Information Systems Management 21 (3) (2004) 41–52.

[22] Donovan Wells, Extreme Programming: A Gentle Introduction. http://www.extremeprogramming.org/ (accessed on: January 2011).

[23] Giuliano Armano, Michele Marchesi, A rapid development process with UML, ACM SIGAPP Applied Computing Review 8 (1) (2000) 4–11.

[24] Luciano Pinto, Ricardo Rosa, Cristiane Pacheco, Christophe Xavier, Raimundo Barreto, Vicente Lucena Jr., Marcus Caxias, Carlos Maurício Figueiredo, On the use of scrum for the management of practical projects in graduate courses, in: Proc. of 39th ASEE/IEEE Frontiers in Education Conference, October 2009, pp. T2C1–T2C6.

[25] Hillel Glazer, Jeff Dalton, David Anderson, Mike Konrad, Sandy Shrum, CMMI or agile: why not embrace both!, Technical Note CMU/SEI-2008-TN-003, Software Engineering Institute, November 2008.

[26] Jauji Shen, Wesley Changchien, Taoyuan Lin, A Petri-net based modeling approach to concurrent software engineering tasks, Journal of Information Science and Engineering 21 (4) (2005) 767–795.

[27] Iraj Mahdavi, Amin Aalaei, Mohammad Paydar, Maghsud Solimanpur, Designing a mathematical model for dynamic cellular manufacturing systems considering production planning and worker assignment, Computers & Mathematics with Applications 60 (4) (2010) 1014–1025.

[28] James Rumbaugh, Ivar Jacobson, Grady Booch, The Unified Modeling Language Reference Manual, Addison Wesley Professional, Indianapolis, Indiana, 2004.

[29] Grady Booch, James Rumbaugh, Ivar Jacobson, The Unified Modeling Language User Guide, second ed., Addison Wesley Professional, Indianapolis, Indiana, 2005.

[30] Juan Guirao, Fernando Pelayo, Jose Valverde, Modeling the dynamics of concurrent computing systems, Computers & Mathematics with Applications 61 (5) (2011) 1402–1406.

[31] Jyhjong Lin, Chunhsiu Yeh, An object-oriented software project management model, The International Journal of Computers and their Applications 10 (4) (2003) 247–262.

[32] Christine Hofmeister, Robert Nord, Dilip Soni, Applied Software Architecture, Addison Wesley Professional, Indianapolis, Indiana, 2000.

[33] Paul Clements, David Garlan, Reed Little, Robert Nord, Judith Stafford, Documenting Software Architectures: Views and Beyond, Addison Wesley Professional, Indianapolis, Indiana, 2002.

[34] Leszek Maciaszek, Requirements Analysis and System Design: Developing Information Systems with UML, Addison Wesley, Reading, Massachusetts, 2001.

[35] Glenn Krasner, Stephen Pope, A cookbook for using the model view controller user interface paradigm in smalltalk-80, Journal of Object-Oriented Programming (1988) 26–49.

[36] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, Reading, Massachusetts, 1995.

[37] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, Pattern-Oriented Software Architecture: A System of Patterns, Wiley & Sons, New York, 1996.

[38] Craig Larman, Applying UML and Patterns, Prentice Hall, New Jersey, 2002.

[39] Microsoft Press, Microsoft Visual C++ Programming with MFC, Microsoft Press, Redmond, Washington, 1995.

[40] George Shepherd, MFC Internals, Addison Wesley Professional, Indianapolis, Indiana, 1996.
[41] Martin Fowler, Patterns of Enterprise Application Architecture, Addison Wesley Professional, Indianapolis, Indiana, 2003.
[42] Leszek Maciaszek, Bruc Lee Liong, Practical Software Engineering: A Case Study Approach, Addison Wesley, Reading, Massachusetts, 2004.
[43] David Kung, Jerry Gao, Pei Hsia, Yasufumi Toyoshima, Cris Chen, A test strategy for object-oriented programs, in: Proc. of 9th IEEE COMPSAC, August 1995, pp. 239–244.
[44] Chienhung Liu, David Kung, Pei Hsia, Chihtung Hsu, Object-based data flow testing for web applications, in: Proc. of 1st IEEE Asia-Pacific Conference on Quality Software, October 2000, pp. 7–16.
[45] Weitek Tsai, Akihiro Saimi, Liguo Yu, Raymond Paul, Scenario-based object-oriented testing framework, in: Proc. of 3rd IEEE Asia-Pacific Conference on Quality Software, November 2003, pp. 410–417.
[46] Clementine Nebut, Franck Fleurey, Yves Le Traon, Jean-Marc Jezequel, Automatic test generation: a use case driven approach, IEEE Transactions on Software Engineering 32 (3) (2006) 140–155.
[47] Kaiyuan Cai, Ping Cao, Zhao Dong, Ke Liu, Mathematical modeling of software reliability testing with imperfect debugging, Computers & Mathematics with Applications 59 (10) (2010) 3245–3285.
[48] Jyhjong Lin, David Kung, Pei Hsia, Object-oriented specification and formal verification of real-time systems, Annals of Software Engineering 2 (1996) 161–198.