IDEA GROUP PUBLISHING

701 E. Chocolate Avenue, Suite 200, Hershey PA 17033-1240, USA
Tel: 717/533-8845; Fax 717/533-8661; URL-http://www.idea-group.com

ITP4893

Formal Support to Incremental Development[1]

Jing Liu
International Insititute for Software Technology, United Nations University, Macau, lj@iist.unu.edu

Zhiming Liu
International Insititute for Software Technology, United Nations University, Macau, lzm@iist.unu.edu
Department of Computer Science, University of Leicester, UK

Xiaoshan Li
Faculty of Science and Technology, University of Macau, Macau, xsl@umac.mo

Jifeng He
International Insititute for Software Technology, United Nations University, Macau, jifeng@iist.unu.edu

ABSTRACT

This paper presents an approach that integrates a formal method with Rational Unified Process (RUP). The intentions are: firstly, unifying different views of UML models that are used in RUP; secondly, supporting effective use of formal method for system specification and reasoning with the iterative and incremental approach in RUP. Our overall aim is to enhance the modeling ability of UML and RUP in preciseness and conciseness. The formal specification notation is based on Hoare and He's Unifying Theories of Programming (UTP).

INTRODUCTION

RUP [6, 7] has recently emerged as a popular software development process. It promotes several best practices, but one stands above the others is the idea of *iterative development*. In the iterative approach of RUP, a project development is organized as a series of short, fixed-length mini-projects called *iterations*; the outcome of each iteration is a tested, integrated, and executable system. Each iteration includes its own requirement analysis, design, implementation, and testing and/or verification activities.

The modeling notation used in RUP is UML [1, 11], that is a de-facto standard modeling language for the development of software in broad application ranges. UML not only supports the early development stages of requirement analysis and specification, but also supports design and implementation [6, 2]. However, like any other semiformal approaches, UML lacks the support of formal semantics for applications with high dependability requirements, and does not meet the IEEE standard for Software Requirements Specification which emphasizes that a good requirement specification should be correct, unambiguous, verifiable, and traceable [13].

In this paper, we propose a method that uses the formal OO notation defined in [5] to describe the models constructed in iterative software development process.

The formal semantics is based on the design calculus proposed in UTP [4]. The character of OO programming is that it includes classes, inheritance, reference types, visibility and dynamic binding. Its refinement calculus [5] supports incremental programming. Because of this, the specification of the models constructed in one iteration can be easily extended or changed in the following iteration.

We use a library system as an example to illustrate the treatment of models created in different cycles of the RUP and how the formal specification and reasoning are effectively used.

A FORMAL NOTATION

Both class declarations and commands follow the notion of *designs* in Hoare and He's Unifying Theories of Programming. All the programming constructs of our language are defined in exactly the same way as their counterparts in the imperative programming languages, in order to make it more accessible to users who are familiar with imperative program design. For the full semantic definition of this notation, refer to the paper [9].

Syntax

In our notation, an object-orient program is of the form *cdecls* · *P*, where *cdecls* is a sequence of variable or class declarations, called the *declaration session*, and *P* is a command. *P* can be understood as the main method of a Java program: *cdecls* := *cdecl* | *cdecls*; *cdecl* where *vdecl* is a *variable declaration*, *cdecl* is a *class declaration* of the following form:

Class N **extends** M{
private U_1 $u_1 = a_1$, … , U_m $u_m = a_m$;
protected V_1 $v_1 = b_1$, … , V_n $v_n = b_n$;
public W_1 $w_1 = c_1$, … , W_k $w_k = c_k$;
method m_1 (**val** T_{11} x_{11}, **res** T_{12} y_{12}) {c_1};
……
m_l (**val** T_{l1} x_{l1}, **res** T_{l2} y_{l2}) {c_l};
}

where

- N and M are names of classes, and *M* is called the direct super class of *N*.
- The section private declares the private attributes of the class, their types and default initial values. similar, the sections pro tected and public for the protected and public attributes.
- The method section declares the methods, their value param eters (**val** *x*) and result parameters (**res** *y*) and their command bodies (c_i).

We will use Java convention to write a class specification, and assume an attribute protected when it is not tagged with **private, protect** or **public**.

This conference paper appears in the book, Innovations Through Information Technology, edited by Mehdi Khosrow-Pour. Copyright © 2004, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Commands

Our language supports typical object oriented programming constructs:

c ::= D a design of the form p H R
| *skip* termination
| *chaos* abort
| **var** T x declaration
| **end** x undeclaration
| c ; c sequence
| $c \lhd b \rhd c$ conditional
| b * c iteration
| *read*(x) read in value
| le := e assignment

where p is a predicate over variables, and R a predicate over state variables and their primed versions, a design $p \vdash R$ is defined to be $p \Rightarrow R$ meaning that if the command start to execute in a state where p holds it will terminate in a state for which R holds, b is a Boolean expression and e is an expression. and le is either a simple variable x or an attribute of an object $le.a$. Command *read*(x) allows us read in a value of x and is defined as:

$$read(x) =_{df} x'e\ type(x)$$

We also define function *find*(x), *add*(x) and *delete*(x) to find, add or delete related values, such as x, in sets or multi objects. In general, an expression can be in one of the following forms:

$$e ::= x \mid \textbf{null} \mid \textbf{new}\ N \mid \textbf{self} \mid e.a \mid f(e) \mid e\ \textbf{is}\ N$$

where the type of null is *Null* which is a class. We can add more expression such as (N)e and type checking e is N, but in this paper, let $\{P_i \mid 1 \leqslant i \leqslant n\}$ be a family of designs. We can define alternation if $\{(b_i \rightarrow P_i) \mid 1 \leqslant i \leqslant n\}$ fi, which selects P_i to execute if its guard is true, or behaves like *chaos* if all the guards are *false*.

SPECIFICATION OF UML MODELS

Model of Requirement

In this paper, a UML model of requirement (RM) consists of a conceptual class model *CM* and a use case model *UM*: *RM = (CM, UM)*. The conceptual model is represented by conceptual class diagram and an invariant [9]. *CM* describes the static view and *UM* the dynamic view of requirement.

Conceptual model

For giving out a formal definition of a class diagram, assume *CN*, *AN* and *AttrN* are three disjoint sets, denoting class names, associations and attributes respectively.

Definition 1. A **conceptual class diagram** is a tuple:

$$\Delta = < C, Ass, Att, \lhd\!\!- >$$

where C is a nonempty finite subset of *CN*, called the classes or concept of Δ. *Ass* is a partial function: $Ass : C\ (AN \nrightarrow PN \nrightarrow PN \nrightarrow C)$. N is the type of natural number and *PN* is the powerset of N. *Att* is a partial function: $Att: C \nrightarrow (AttrN \nrightarrow \tau)$, where τ is a set of variables that define the properties of the objects of the class. We use $C.a : T$ to denote $Att(C)(a) = T$, and call an attribute of C and T the type of a. $\lhd\!\!- \subseteq C \times C$ is the direct generalization relation between classes.

Definition 2. A **conceptual model** is $CM = <\Delta, Inv>$, where Δ denotes a conceptual class diagram and *Inv* is state constraint over Δ.

The state of a class diagram is an object diagram that defines a snapshot of the system with current existing objects, values of their attributes and links among the objects [8, 9]. A state property φ of a conceptual model can be reasoned about by proving the implication $\vartheta \wedge Inv \Rightarrow \varphi$ in the relational calculus. It also allows us to define transformation between conceptual diagrams that preserve state constraint.

Use Case Model

System executes a system operation when it is called by an actor. So we model a use case by defining system operations as a *command*.

Then the model of the system requirements can be specified as the transition system $S =_{df} <CM, Inv, UM, Init>$ [10, 9]. With the formal OO notation, a model $RM = (CM, UM)$ of requirements is specified as a program $cdecls \cdot P$, in which *cdecls* is a sequence of class declarations that formalize *CM* and P is the main method that specifies the use cases *UM*. The formal semantics of this program defined in [5] captures both syntactic and semantic consistency between *CM* and *UM*. This formalization tightly couples together the static view and the dynamic view of an UML model of requirement. Any inconsistency between *CM* and *UM*, such as the access to a variable whose type has not been declared in class diagram or the access to a variable in a method that is not visible in the class of the method, will lead the program $cdecls \cdot P$ behaves as *chaos*. In our framework, we can fix an inconsistency by refining the program.

A library system

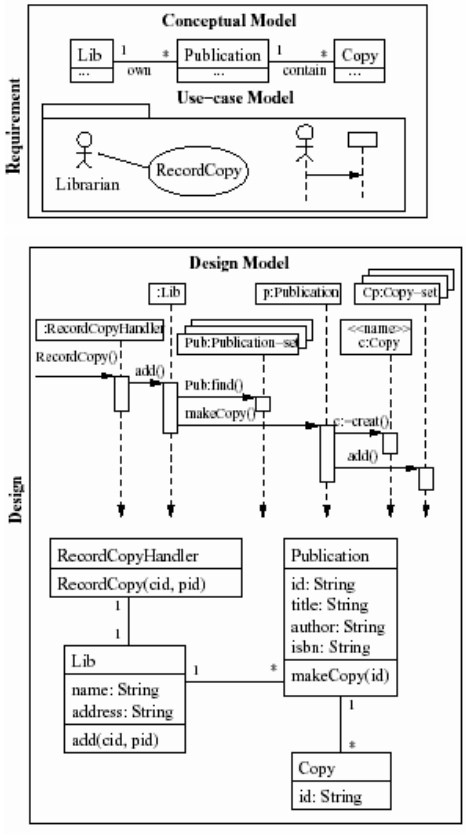
Now consider a simple library system. There are various kinds of publications in the library and each kind of publication has some copies. Only members registered to the library are allowed to borrow copies of publications. Reservation service is also provided to members.

Conceptual model

In the first iteration we only concerned one use case *RecordCopy* which records copies of some publications to the library, as shown in Figure 1. We can specify it as the following class declaration section and denote this declaration section as $\mathbf{CM_1}$.

Class *Lib*{ *String name, String address, String id*};
Class *Publication*{ *String id, String title, String author, String isbn*};
Class *Copy*{ *String id*};
Class *Owns*{ *Lib lib; Publication p*};// **Association Classes**

Figure 1: Models in the First Iteration



Copyright © 2004, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Class *Contains* { *Publication p; Copy c*};// **Association Classes**

We assume the system initial state is: *PLib lib* = {*lib*}; *PPublication Pub* = ∅; *PCopy Cp* = ∅; *POwns Ow* = ∅; and *PContains Co* = ∅.

Use case model

An identification of the use case is important for the creation of the conceptual model. However, the formal specification of the use cases depends on the specification of the conceptual model. We provide a *canonical form* of a use case specification by introducing a class of use-case handler, which encapsulates the conceptual model [8]. Considering the use case *RecordCopy* that adds a new copy of an *existing* publication to the library. We specify this use case by introducing a class *RecordCopyHandler*:

CM$_1$; *// import the conceptual model*
Class *RecordCopyHandler* {
  Method *RecordC*(**val**(*String cid; String pid*)){**var** *Copy c*
    Ep ε *Pub* • *p.id* = *pid* H
    c := **New** *Copy*(*cid*);
    Cp := *Cp* Y {*c*}; *Co* := *Co* Y {< *p, c* >}; **end** *c, p*}}
Use Case *RecordCopy* :: {
  var *RecordCopyHandler h, String cid, String pid*
  h = **New** *RecordCopyHandler()*;
  read (*cid, pid*) ;
  h.RecordC (*cid, pid*); **end** *h, cid, pid*}

The **main** method is then specified as:

main( ){**var** *Bool stop, Services s, stop* : = *false*;
  while *!stop* **do** {*read*(s);
    if{(*s* = "*RecordCopy*") → *RecordCopy*} **fi**;
    read(*stop*)}; **end** *stop, s*}

where the type *Services* denotes the services that the library system will provide. Note that a UML conceptual model also determines some state invariants, such as there is only one *Lib* instance. The use cases need to be checked to preserve these invariants [8].

Design Model

A UML model of design *DM* = (*DCM, SD*) consisting of a *design class diagram DCM* and a family of *object sequence diagrams SD*. Similar to the formalization of a conceptual class diagram, we formalize *DCM* as a declaration section $cdecls_d$. Classes in this declaration section now have methods and a method of a class may call methods of other classes. Therefore, the specification of these methods describes the object interactions. However, methods are activated by commands in the main program P_d. Therefore, a UML model of design (*DCM, SD*) is only specified as a declaration section $cdecls_d$. The consistency between the class model *DCM* and the object sequences diagrams *SD* is captured by the semantics of $cdecls_d$ and the semantics of method calls in the formal OO notation. The *correctness* of the design model (*DCM, SD*) *w.r.t* the requirement model *RM* = (*CM, UM*) is captured by the *declaration refinement* relation $cdecls \sqsubseteq cdecls_d$ that is defined in [5]. The main method in the design is almost the same as that in the requirement model.

A method is described as (*m*(**val** T_1 *x*, **res** T_2 *y*), *c*), which have a name *m*, signature *m*(**val** T_1 *x*, **res** T_2 *y*) and a body that is a command *c*, where **val** T_1 *x* is a list of value parameters and **res** T_2 *y* a list of result parameters.

Definition 3. A **design class diagram** is a tuple:

$$\Re = <C, Ass, Att, \lhd\!\!-, Met>$$

where *Met* is a mapping from *C* to a set of methods.

An association in a design class diagram is directed and can be represented as an attribute of the source class. As the multiplicity of the target role in an association is more than one in general, such that an attribute is of type of the powerset of the target class. However, when the multiplicity is {1} or {0, 1}, we only represent it as an attribute with the class as its type. Therefore, a class diagram can be formally specified as a sequential composition of class declarations of the form:

Class *N* **extend** *M* {
  T a ;
  D ass ; // *D* is either a class *C* or its powerset *PC* for
    // each association *ass* from *N* to *C*
  Method m_1(**val** T_{11} x_{11}, **res** T_{12} y_{12}) {c_1};
    .....
    m_k(**val** T_{k1} x_{k1}, **res** T_{k2} y_{k2}) {c_k};
}

A sequence diagram consists of *objects* and *messages* that describe how the objects communicate. An interaction occurs when one object invoke a method of another.

The library system

Assume for each type *PC*, methods *add*( ), delete( ), and *find*( ) are declared for adding, deleting and finding an object of *C* in a set. The object sequence diagram and its corresponding design class diagram in Figure 1 are specified by the following program. Assume classes *Pub*, *Cp* and *M* have declared these two methods. The design model **DM$_1$** is constructed as:

Class *Lib* {*String name, String address*;
    PPublication Pub; // newly added
    Method *add*(**val** (*String cid, String pid*)) {
    var *Publication p*;
    p = *Pub.find*(*pid*); *p. makeCopy*(*cid*); **end** *p*}}
Class *Publication* { *String id, String title, String author, String isbn, Copy-Set Cp;*
  Method *makeCopy* (**val** *String id)* {
  var *Copy c*; *c* := **New** *Copy(id)*; *Cp* : = *Cp.add*(*c*) ;
  end *c*}}
Class *Copy* {*String id* }
Class *RecordCopyHandler* { *Publication-Set Pub, Copy-Set Cp, Contain-Set Co*;
Method *RecordCopy*(**val** (*String cid, String pid*)){
    Lib . add (*cid, pid*) **end** } }

The correctness of the design is captured by the refinement relation defined in [5].

Iterative and Incremental Development

RUP is a use-case driven, iterative and incremental software development process. In each iteration, several use cases are chosen and added to the requirement model RM_1 = (CM_1, UM_1) got in the previous iteration. Based on the previous analysis and design results DM_1 = (DCM_1, SD_1), a new requirement model RM_2 and design model DM_1 are established. RM_2 can be obtained by enlarging RM_1 with newly added use cases. And then a new design model DM_2 can be obtained from refining RM_2 and DM_1. Such a refinement is usually called superimposing the design for the new use cases to the design model DM_1.

In the next iteration the output of the previous iteration $cdecls_1 \cdot P_1$ will be reused in the next iteration $cdecls_2 \cdot P_2$, and to preserve the correctness of the previous iteration, we require $cdecls_1 \cdot P_1 \sqsubseteq cdecls_2 \cdot P_2$. In particular, we require the declaration refinements to hold:

$$cdecls_1 \sqsubseteq cdecls, \quad cdecl_{d1} \sqsubseteq cdecls_{d2}$$

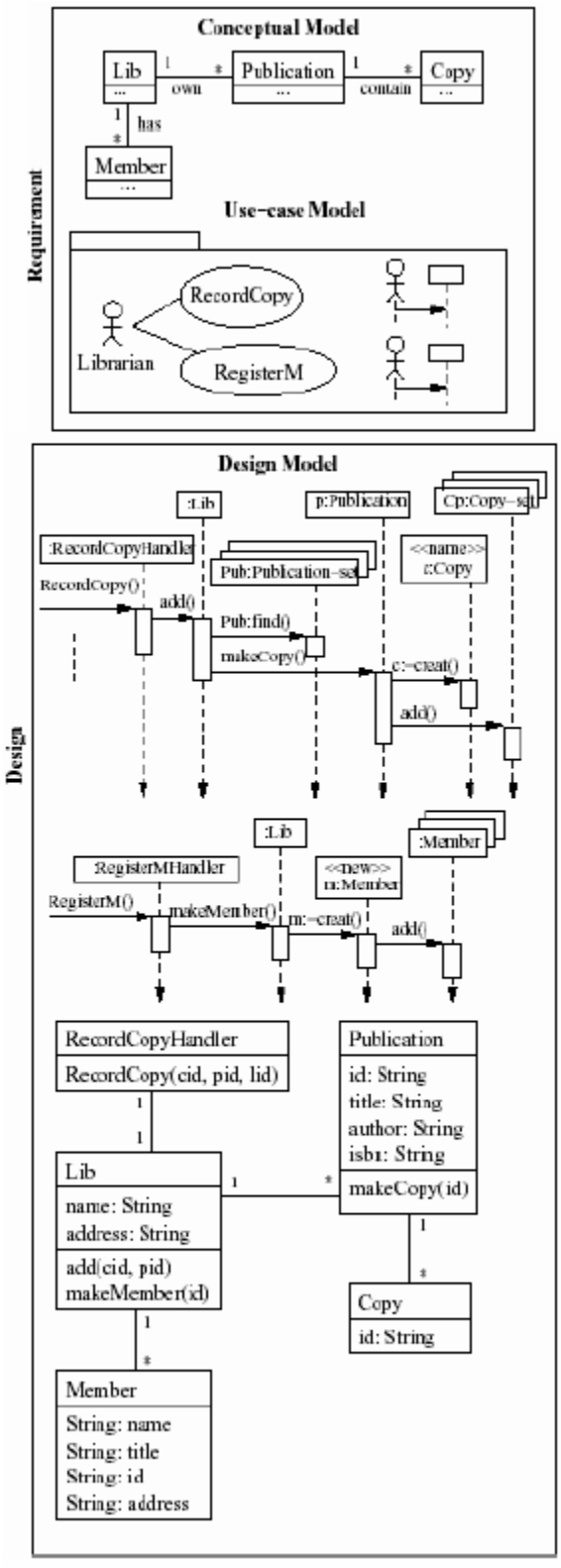
It means that conceptual and design class diagrams (declarations) of the new iteration are refinement of those in the previous iteration respectively.

We use the library system to demonstrate the process. Assume RM_1 and DM_1 denote the requirement model and design model in the first iteration. RM_2 and DM_2 denote the requirement model and design model in the second iteration.

In the second iteration, we expand RM_1 by adding use case *RegisterM*, which is used to register a member to the library, as shown in **Figure 2**.

Copyright © 2004, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Figure 2: Models in the Second Interation



The conceptual model $\mathbf{CM}_1$ is extended to $\mathbf{CM}_2$**:**

$\mathbf{CM}_1$**;** //Reuse conceptual model
Class *Member*{*String name, String title, String id, String address*};
Class *Has* {*Lib lib, Member m*};

Use case The use-case handler for *RegisterM* is then specified as: let *StringList* be the type of the list of strings of the attributes of

Member : *StringList l* - {*String name* × *String title* × *String id* × *String address*}
Class *RegisterMHandler* {
Method *RegisterM* (**val** (*StringList l*)){
$\neg\exists m$ e M• $m.id = mid \vdash$
m: = **New** *Member*(*l*)
$^{\frown}M' = M \cup \{m\}^{\frown}$ *has'* = *has* $\cup$ {< *lib, m* >}}}
Use Case *RegisterM* :: { **var** *RegisterMHandler mh, StringList l*;
mh := **New** *RegisterMHandler*( ); *read*(*l*);
mh.RegisterM(*l*); **end** *mh, l*}

Like the use case *record copy*, use case *RegisterM* can be verified or tested alone. Then the main program will be enlarged into the following one by adding *RegisterM* as a service:

main(){ **var** *Bool stop, Services s;*
stop := *true*;
while *!stop* **do** {*read*(*s*);
if {(*s* = *"RecordCopy"*) → *RecordCopy*} **fi**;
if {(*s*= *"RegisterM"*) → *RegisterM*} **fi**;
read(*stop*)}}

Design model, in turn, is expanded as following. The interaction diagrams and associated class diagram are shown in Figure 2.

Class *Lib* {String name, String address, Publication-Set Pub, Member-Set M; // newly added
Method *add* (**val** (String cid, String pid)) {
var *Publication p*;
p = *Pub.find*(*pid*);
p.makeCopy(cid);
end *p*}
makeMember(**val** *StringList l*){**var** *Member m*;
m = **New** *Member*(*l*); *M* := *M.add*(*m*); **end** *m*}

}
$\mathbf{CM}_2$; $\mathbf{CM}_1$; // import newly added classes
$\mathbf{DM}_1$; *//import preceding design model*
Class *RegisterMHandler* {*Memeber-Set M, Has-Set has*;
Method *RegisterM*(**val** *StringList l*){ **var** *Member m*;
m := **New** *Member(l)*; *M* := *M.add(m)*; **end** *m*}}
Use Case *RegisterM // unchanged*

Without any change to the conceptual model $\mathbf{CM}_2$, we can also specify and design use cases *SerchMember*, *SerchPublication* and *SerchCopy*. The software system is then enlarged iteration by iteration.

CONCLUSION

We have presented a formal method to support the iterative and incremental software development. The method is based on a formal model of specification and refinement of object-oriented systems in [5]. The important nature of this method is that each iteration only considers a small part of the system functionality. Instead of using a traditional compositional approach, we decompose the system informally according to use cases. Therefore, we can work with a small model at a time. Then the formal model obtained in each iteration is composed to form a larger system.

Copyright © 2004, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Our primary motivation is to enhance UML and RUP with formal method and unifying views of UML models. This can be done by defining precise description of the requirement models and design models, as well as developing mechanisms, so that developers can rigorously analyze the models.

There is a lot of existing work on formalization of UML, but it is difficult to give a relatively full account of it, because of the space limitation. However, the major distinct nature of our work is that it concerns on the consistent relations (both horizontal and vertical) among the different models rather than the formalization of a particular model. Another advantage of our work is that the notation is Java-like and easy to use. Further more the semantics is relational and based on first order logic that are easy to understand.

Further work includes the extension of this method to component-based software development [12, 14, 2, 3].

REFERENCES

[1] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modelling Language User Guide*. Addison-Wesley, 1999.

[2] D. D'Souza and A.Wills. *Objects, Components and Framework with UML: The Catalysis Approach*. Addison-Wesley, 1998.

[3] G. Heineman and W. Councill. *Component-Based Software Engineering, Putting the Pieces Together*. Addison-Wesley.

[4] CAR. Hoare, and J. He, Unifying Theories of Programming, Prince-Hall International, 1998

[5] J. He, Z. Liu, and X. Li. Towards a Refinement Calculus for Object-Oriented Systems. In *Proc. ICCI02, Alberta, Canada*. IEEE Computer Society, 2002.

[6] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[7] P. Kruchten. *The Rational Unified Process – An Introduction. 2nd edition*.

[8] Z. Liu, J. He, X. Li and J. Liu. A Relational Model for Object-Oriented Analysis with UML. To appear in Proc. ICFEM03, Singapore, 4-7 November 2003.

[9] Z. Liu, X. Li, and J. He, Using Transition Systems to Unify UML Models, Lecture Notes in Computer Science, Springer LNCS 2495, 2002.

[10] Z. Mana and A. Pnueli. The temporal framework for concurrent programs. In R.S. Boyer and J.S. Moore, editors, The Correctness Program in Computer Science, Academic Press, 1981.

[11] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modelling Language Reference Manual*. Addison-Wesley, 1999.

[12] C. Szyperski. Component Software: Beyond Object-Oriented Programming. Addison-Wesley.

[13] D. R. Windle and L. R. Abreo, Software Requirements Using the Unified Process, page 7, Prentice Hall, 2003.

[14] L. Barbosa and M. Sun, Generic Components, Proceedings of First APPSEM II Workshop, 2003.

FOOTNOTES

[1] The work is partly supported by research grant 02104 MoE and 973 project 2002CB312000 of MoST and Natural Science Foundation of China 60173030.

Copyright © 2004, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

0 more pages are available in the full version of this document, which may be purchased using the "Add to Cart" button on the publisher's webpage:
www.igi-global.com/proceeding-paper/formal-support-incremental-development/32303

Related Content

Online Information Retrieval Systems Trending From Evolutionary to Revolutionary Approach
Zahid Ashraf Wani and Huma Shafiq (2018). *Encyclopedia of Information Science and Technology, Fourth Edition (pp. 4535-4547).*
www.irma-international.org/chapter/online-information-retrieval-systems-trending-from-evolutionary-to-revolutionary-approach/184161

8-Bit Quantizer for Chaotic Generator With Reduced Hardware Complexity
Zamarrud and Muhammed Izharuddin (2018). *International Journal of Rough Sets and Data Analysis (pp. 55-70).*
www.irma-international.org/article/8-bit-quantizer-for-chaotic-generator-with-reduced-hardware-complexity/206877

Self-Adaptive Differential Evolution Algorithms for Wireless Communications and the Antenna and Microwave Design Problems
Sotirios K. Goudos (2015). *Encyclopedia of Information Science and Technology, Third Edition (pp. 5754-5766).*
www.irma-international.org/chapter/self-adaptive-differential-evolution-algorithms-for-wireless-communications-and-the-antenna-and-microwave-design-problems/113030

A Comprehensive Survey on Face Image Analysis
Yu-Jin Zhang (2015). *Encyclopedia of Information Science and Technology, Third Edition (pp. 491-500).*
www.irma-international.org/chapter/a-comprehensive-survey-on-face-image-analysis/112362

The Bees Algorithm as a Biologically Inspired Optimisation Method
D.T. Pham and M. Castellani (2015). *Encyclopedia of Information Science and Technology, Third Edition (pp. 285-294).*
www.irma-international.org/chapter/the-bees-algorithm-as-a-biologically-inspired-optimisation-method/112336