Enhancing Software Reuse through Application-level Component Approach

Jin Guojie, Yin Baolin, Zhao Qiyang
Chinese State Key Laboratory of Software Developing Environment
Beihang University
Beijing, China, 100191
Email: {jinguojie, yin, zhaoqiyang}@nlsde.buaa.edu.cn

***Abstract*—Current component reusability is not as high as previously expected. Although third-party component providers are in present, large quantity of reduplicative programming effort is still needed in system development process. As conventional component technologies are not flexible enough to deal with requirement diversity and variability, a new type of Application-Level Component (ALC) approach is proposed. The functional granularity of ALC is larger than that of previous components, thus lowering the effort for reusing a component. By separating the stable and instable part of domain requirement, a collection of stable requirement elements can be summarized and implemented by ALCs. The instable part can then be described with a formal language according to the differences in various user cases. A novel reuse process of "selection and description" is established. The description language covers overall aspects of application system requirements, including user interface, computation logic, and database access. The description content for a system is parsed and executed by ALC to fulfill corresponding requirement. By providing different description contents, ALC can be reused in environments full of differences and changes. Evaluations reveal that the reusability of ALC is enhanced to a higher degree of 92.5~95.7%.**

***Index Terms*—Application-level Component; Reusability; Component Granularity; Requirement Description Language; UI patterns**

I. Overview

Component technology plays an important role in software reuse research. As expected in Component-Based Software Development (CBSD) methodology, one can develop a system simply by selecting and assembling current-existing components. However, this goal is far from reality. In real development environments, large amount of reduplicated programming effort is still needed, because of the lack of appropriate components [1]. In Ref. [2], an investigation is undertook in 2005 aiming at 25 software projects form NASA, and the average reusability is measured at a degree of just 32%. Peer results can also be found in Ref. [3], which investigates the current situation of software reusability in China. The result is no better than the former. A significant conclusion is that, different styles of components have different degrees of reusability. The fundamental computation components have the highest degree of 99.5%, but the business-oriented UI components only have a degree of 8.4%. In average, the reusability reaches only 27.7%, which means almost 3/4 components in systems are not able to be implemented by direct reuse and are needed to be created by the developers themselves.

There can be many reasons for such phenomena. An important point is observed that, components are always prevented from being reused because of the differences between its predefined function and the system requirements. In this paper, we focus on two natures of requirement which frequently cause such differences. Those are *diversity* and *variability*. According to the nature of diversity, different organizations are running various business rules. It's theoretically impossible to provide a finite component repository covering all the business rules around the world. Even in the relatively matured domains, like ERP and OA, there are unavoidable differences which are hard to be unified in a foreseeable period. The second nature means that the business rules of an organization is changing all the time, to cope with the external influences originated by variable economic circumstance, new marketing trends, etc. Once the requirement changes, the software system must change its function in parallel to fulfill new business rules.

Research efforts are not made enough to provide flexibility in software components, resulting in the lack of power to handle requirement diversity and variability. More sufficient reasons can be shown by comparing the following three typical component technologies.

1. *UML/Catalyst method* [4]. UML is a representation tool extensively incorporated in Object-Oriented Analysis & Development (OOA/OOD) processes like Catalyst. The design processes all start from the observation towards the functional requirements of an object system. Since the system is likely to serve the business rules of a dedicated organization, it is usually hard to directly reuse parts of the system for other organizations. As requirement changes, the design documents and the components implemented are all forced to be modified. This prevents the components to be "reused without modification".

2. *Feature-Oriented Component Model* (FOCM) [5]. FOCM is a typical method for domain requirement modeling. With FOCM, requirements of different application systems can be combined to one model via a

doi:10.4304/jsw.6.3.374-385

series of modeling elements which represents differences and variations. Compared with UML, FOCM effectively extends the knowledge scale of the modeling tool to a business domain, which causes the components to be able to be reused in different organizations instead of just one organization. But, problems remain that it does not provide flexibility to handle future requirement changes. Using FOCM, designers have to endlessly catch the requirement changes, but always lag behind requirement changes.

3. *Framework-based Development* (FBD) [6]. FBD attempts to enlarge the representation ability towards requirement differences at the level of software architecture. FBD utilizes FOCM as its description tool for software components, so it has the same problem with FOCM, e.g., lacking of ability to support unforeseen requirements probably emerging in the future.

As current component technologies do not provide effective supports for requirement differences and changes, third-party components cannot be reused straightforwardly as expected by researchers. To reuse a component, modification effort makes up most of the cost to eliminate the difference between its predefined function and actual object requirement. Furthermore, the modification frequency is judged by different stability degrees of requirement elements. Observation shows that the components with the highest reusability lie in the levels adjacent to computer implementation [3]. Such components include conventional computing routines, basic UI controls, etc. The components fall into this category usually take up a smaller functional granularity compared with the high-level business-oriented components assembling the ultimate system, and their abstract level is lower than that of the latter. When one needs to create an ERP or OA system, and there is probably lack of appropriate components supporting his specific requirement (this is a frequently-seen phenomena in application development), he has no choice but to create his own business-oriented components by assembling the small-granularity components with programming codes. Such is a kind of "selection and programming" reuse process. Since the programming codes are usually designed for solving his private needs instead of being reused in public situations, they form the non-reusable part of software systems. The reusability cannot be enhanced until new technology tackles this problem.

A novel design process named Application-level Component (ALC) is proposed to enhance software reusability. By utilizing new design methods, ALC is responsible for implementing the stable elements and construction mechanisms of domain requirements. Customization ability is also equipped to support future requirement variations. Compared with conventional components, ALC takes up a higher degree of reusable functional granularity, thus lowering the effort for dealing with requirement diversity and variability.

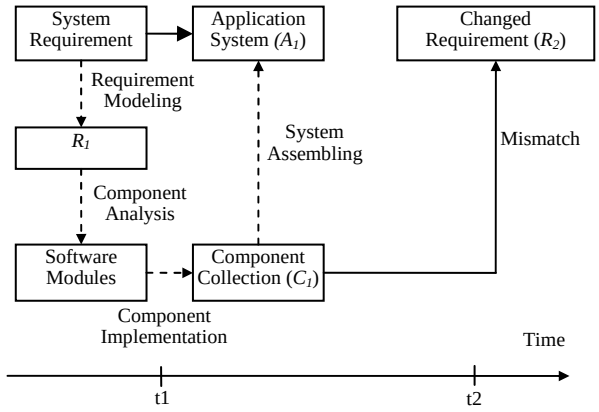


Figure 1. Problem of Existing CBSD Processes

II. Problem in Existing CBSD Processes

Being focused on the common natures, different kinds of current CBSD processes can be summarized as a unified process, as shown in Fig. 1. In Ref. [4], [5] and [6], different kinds of design processes based on UML, FOCM or FBM are proposed. They all comply with the unified process discussed as follows. A completed process cycle consists of four steps. First is *Requirement Modeling*, in which developers investigates the requirement of a system or an application domain, and define it as R_1; Second is *Component Analysis*, where R_1 is separated into a series of functional modules, and specifications for each module's function and interfaces are defined; Third is *Component Implementation*, where all the components are developed according to the specifications, resulting in a component repository C_1. In the final step of *System Assembling*, C_1 can then be used to assemble a system A_1, the function of which is exactly equivalent to the original requirement.

For two reasons, the reusability of components is limited by such a process.

1. From the perspective of time, the requirements of the components are relatively stable instead of absolutely stable. As the system requirement will change forever, R_1 is only a snapshot at a dedicated time t_1. Suppose the requirement changes to R_2 when it comes to t_2. Since $R_1 \neq R_2$, it cannot be guaranteed theoretically that the components designed according to R_1 can straightforwardly be reused in the new situation. So, some items in C_1 need to be modified, and new items need to be appended into C_1. As time goes, this appears at any time and never stops. One can never provide a component repository that can fit all the requirements in the future.

2. From the perspective of organization domain, the requirements of the components are relatively complete instead of absolutely complete. Because of the natural limitation of human's far-sight and the economical constraints of development cost, any component repository can only cover a part of the whole organization space. By contrast with the openness and diversity of requirement space, the function of any component repository is incomplete.

As a result, only when the predefined function fits exactly in the object requirement can the components be reused. This only reflects a small portion of the real situations in system development. It also brings a side-effect, that the reusability of components is decided by the stableness of different requirement levels [4]. In the level adjacent to computer implementation, the stableness of requirement elements gains a high degree. The components at this level, including mathematical routines and UI controls, form the most frequently reused part of software resource in reality. But, as the abstract level rises, the probability of requirement changes increases proportionately due to the enlargement of business scale and complexity, and the reusability gradually drops to a low degree. Just because of this, the business-oriented components residing at this level, including ERP and OA components, gain a low reuse degree of only 32%, as Ref. [2] claimed.

As far as granularity is concerned, the business-oriented components are usually in larger scale than the low level components. When developing systems, great amount of effort is needed to program assembling codes for grouping small components into larger business-oriented components. In actual circumstances, the total efforts made to reuse the components may even exceed the benefit gained through reuse. Meanwhile, as such effort is usually specific to cope with private requirement and not conducted according to the principle of "design for reuse", such single-time efforts are reduplicated in each time of development. Our goal is to transform such efforts into once-for-all software resources, so as to effectively enhance the potential of reuse.

III. Application-level Component Approach

The starting point of Application-level Component (ALC) approach is to enlarge the functional scale of reusable software module, while at the same time decreasing the effort for providing business-oriented components. In this way, the reusability of software resources can be enhanced. To enlarge the scale of reusable software module, ALC approach investigates the commonness in high-level business-oriented components. Then, such commonness is undertaken by ALC components, which cover a larger part of reusable constitutions in systems. As the result of the approach, the scale of reusable components is finally enlarged to the peer level of business-oriented user cases.

Definition 1. Application-level Component (ALC). Application-level component is a kind of reusable software module, the functional scale of which is equivalent with that of business- oriented user cases.

As the definition claims, ALC can provide reusable modules at a larger granularity and higher abstract level than conventional components. These are exactly the granularity and level of business-oriented components that are in short of in systems like ERP and OA. As the reusable functional scale increases, the effort for reuse could definitely be lowered. The key point is how to increase the reusability of components at this granularity and level. For two reasons, the feasibility of the approach can be guaranteed.

1. Although the requirements differ in various organizations and at different time, there exists a finite set of stable elements and construction mechanism among them. These form the commonness of different system instances within a domain. The set of elements includes the three aspects of system constitutions, e.g., UI elements, computing logic elements, and data access elements. Such elements reside at a high abstract level, e.g., the requirement level. The abstract degree is far higher than that of the implementation level, where the fundamental small components exist. Based on the stable elements, every system instance can be constructed via the common construction mechanism, by conducting different behaviors of elements and communications among the elements. This ensures the feasibility that a finite set of common features can be summarized from present system instances, and can be utilized to construct all the system requirements in the future.

2. Once the set of common features are summarized, ALC components can be developed to implement the functions of requirement elements. They actually take on the commonness of requirements. When constructing a system, the only effort needed is to provide the information of the way that a dedicated system is constructed from the requirement elements. Since the set of common elements is finite, a formal language can be established to describe the construction information. An instance described by the language can be parsed and automatically executed by the ALC components to fulfill the corresponding requirement. In this way, the functions of ALC components are aimed at requirement level, and their granularity is enlarged to capsule a set of basic elements, which are capable of constructing a complete business-oriented user case, or even a whole system.

According to the process, the requirement of a system is separated into two parts. One part contains the stable and common requirement elements, which are undertaken by ALCs; the other part is described in a kind of Requirement Description Language (RDL), which represents the dedicated construction way of a system. A standard ALC, along with a piece of description instance, make an ALOC, which has the granularity of business-oriented user cases and can be assembled in the final system.

Definition 2. Application-level Object Component (ALC). Application-level object component is a standard *ALC* as well as a requirement description instance *DESC* which represents the function of a business-oriented user case. So

$$ALOC ::= \langle ALC, DESC \rangle$$

In an *ALOC*, the *ALC* part is a component directly reused; the *DESC* part is necessary for describing the differences among various systems, including the activities that the elements performed, and the communication actions occurred among elements.

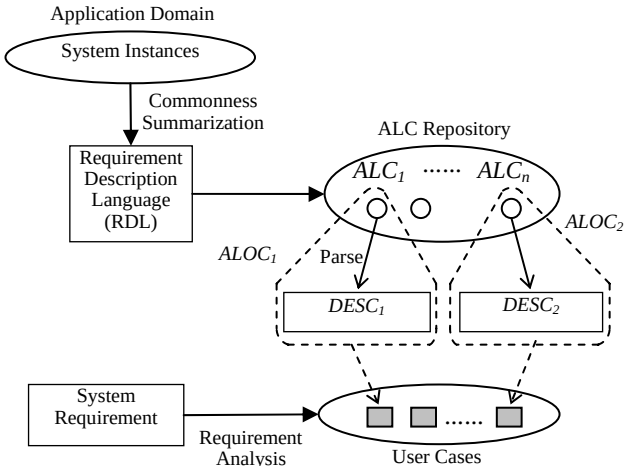


Figure 2. Reuse process of ALC

A novel reuse process of "selection and description" is then formed and shown in Fig. 2. The RDL uses predefined requirement elements as its syntax symbols. Each element has a set of attributes and actions representing different business semantics. A user case can then be described by a sequence of element behaviors and communications. The ALCs are responsible for parsing the RDL syntax and transforming the element behaviors into software semantics. The whole description instance can thus be mapped into computer implementation.

IV. Analysis of ALC's Reusability

Enhanced component reusability can be gained from the new ALC-style "selection and description" process. Compared with the conventional reuse style of "selection and programming", there is no need for reduplicated programming work In ALC-style process. Instead, only the requirement description content is needed to be provided to create business-oriented components. The ALC components have the same granularity with object user cases, which is larger than that of conventional components, thus reducing the scale of additional information needed for use. Furthermore, as RDL is oriented at the business level, it is much more concise than traditional programming language, and requires smaller description scale when describing the same requirement. These simultaneously form the basis of ALC's reusability enhancement effect.

Here we analysis the reusability of ALC with a semi-quantitative method. To compare the degrees induced by different design processes, a unified reusability criterion is defined.

Definition 3. Component Reusability. $r(U)$refers to the proportion of functional scale reused from components in a user case U,

$$r(U) = \frac{E_c}{E_u} * 100\% \quad (1)$$

where E_c is the functional scale reused, and E_u is the whole functional scale of user case U.

When creating a business-oriented component responsible for a dedicated user case, benefits can be gained by reusing some appropriate components. Apart from the functions reused, other functions exist which cannot be borrowed from existing components, due to the nature of requirement diversity and variability. Therefore, unavoidable efforts should be made to provide remain information supporting those functions. For these two parts, suppose E_c is the function scale reused from components and E_e is the scale of additional information provided, so the scale of overall functions E_u is the sum of these two parts,

$$E_u = E_c + E_e \quad (2)$$

From (1) and (2),

$$r(U) = \frac{E_c}{E_c + E_e} * 100\% \quad (3)$$

The nature of reusability is revealed in (3) that, the more effort we need to create a user case, the lower the reusability of the components is; on the other hand, keeping the effort E_e at a certain level, the larger scale of functions can be reused from existing components, the higher the reusability is gained.

For a whole system A, Eq. (3) can be slightly extended to measure the overall reusability reflected in the system. When calculating $r(A)$, E_c is calculated by summing up the function scale that is reused in each user case, and E_e is the sum of the function scale of the remaining portion. Considering the apparent common nature between $r(U)$ and $r(A)$, we will focus on $r(U)$ for simplicity in the following discussions.

In conventional component design processes shown in Fig. 1, assuming there exists a component C which exactly matches the requirement of user case U, only a small quantity of customization effort is needed for reusing the component. According to Eq. (3), E_e can be ignored against E_c, therefore $r(U)$ reaches the theoretical upper bound of 100%.

However, things are different in reality. Due to the enormous diversity of user requirements, there is a great chance that such "perfect" component cannot be found in the market. To avoid developing a component from scratch, one should usually undertake modification work based on an "approximately suitable" component. In general, two kinds of modifications can be performed. First, developers may directly modify some business-oriented components which are distributed along with their source codes; second, developers can create new business-oriented components by reassembling some low-level fundamental components. According to Eq. (3), we get

$$E_e = E_{modification} + E_{assembling} \quad (4)$$

e.g., the reuse effort E_e consists of two parts, where $E_{modification}$ stands for the effort for modification work, and $E_{assembling}$ stands for the effort for assembling existing low-level components. The reusability of conventional components can be defined by combining Eq. (3) and (4).

Definition 4. Reusability of Traditional Component (RT). The reusability of components through conventional processes is measured as $rt(U)$, which is defined as

$$rt(U) = \frac{E_c}{E_c + \left(E_{modification} + E_{assembling}\right)} * 100\% \quad (5)$$

For ALC approach, the measurement method differs due to the different reuse process utilized. Based on the principles adopted by ALC approach, all the common elements and constructing mechanism in a domain requirement are supported by a collection of ALC components *C*. Such a collection *C* covers the requirements of current and future systems in the domain. For any user case *U* in a system *A*, there exists a couple of components in *C* which provide some functions capable of being reused in *C*. Benefited from the reuse style of "selection and description", only the effort of describing the requirement instead of modifying the component is needed. As a result, the reusability of ALC components can be defined.

Definition 5. Reusability of Application-level Component (RA). The reusability of components defined and reused through ALC process is measured as $ra(U)$, which is defined as

$$ra(U) = \frac{E_c}{E_c + E_{description}} * 100\% \quad (6)$$

where $E_{description}$ is the effort for describing the requirements of object-oriented components corresponding to each user case.

By comparing Eq. (5) with (6), it can be seen that the difference between two kinds of reusability lies only in $E_{modification} + E_{assembling}$ and $E_{description}$. In the next analysis, it is proved that the latter is smaller than the former because of three qualitative reasons.

1. *The technical complexity of ALC-style reuse process is lower*. In the conventional reuse process of "selection and modification", developers should be familiar with the technical details of both existing components and current requirements; the modification work is primarily done through a long-period streamline consisting requirement planning, implementation design, code programming, verification, and maintaining. Whereas in ALC process, the developing streamline contains only one stage of "describing"; the style of developing work is changed into a more concise way of "what you describe is what you get". The time cost of requirement description work is significantly lower than that of conventional processes.

2. *The requirement of developers' technical skill is lowered in ALC process*. In conventional reuse processes, developers perform modification activities using some kind of traditional programming language. In ALC process, the language utilized changes to RDL. As RDL is a tool aiming at straightforwardly describing ultimate requirements, such high-level language is easier for developers to comprehend and manipulate than general-purpose programming languages. In our experiences, even some business users can be taught using RDL to create relatively preliminary software tools independently. Furthermore, there can be fewer chances for developers to misunderstand the users' requirements by using RDL as a communication tool, which is more precise and can be understood by both sides. With this more effective language, development task can always be accomplished in a shorter period.

3. *The scale of information provided for reuse is reduced by ALC approach*. RDL is characterized by its problem-oriented and business-oriented features. It aims at a higher abstract level than programming languages. As a result, when implementing the same requirement of a user case, small description scale is needed by using RDL than traditional programming languages. As the scale of description information decreases, the reuse effort is accordingly lowered. The following two examples are presented to aid this opinion.

Example 1. A basic instruction of "`SUM(array A)`" is provided in RDL, which implements the semantic of "counting the sum of elements within an array *A*". With programming languages, at least a looping structure should be coded to represent the execution details of the traversing logic, which may occupy approximately several code lines.

Example 2. Consider more complex requirements emerge in user interfaces. It is well recognized that developers are obliged to make enormous effort to deal with the complicated coupling logic among small UI controls and dialogues. References claim that 70% scale of code lines in a system are occupied by UI requirements [7]. Owing to the UI description language discussed in Section 5, such UI requirements can be expressed in a much more concise style, and considerable scale of description lines is saved.

The above three aspects of analysis jointly give evidences that, when developing a business-oriented component for a dedicated user case, the ALC process reduces the effort for reuse.

$$E_{description} < E_{modification} + E_{composition} \quad (7)$$

An essential conclusion can be drawn by combining Eq. (5), (6), (7) that, the reusability of ALC components is enhanced.

Conclusion 1. The advantage of reusability enhancement through ALC approach. For the construction of a user case *U*, the reusability gained in ALC process is higher than that in conventional processes.

$$ra(U) > rt(U) \quad (8)$$

Considering the common nature between a user case and a system, Eq. (8) can be extended to form the next conclusion applied to system scale.

Conclusion 2. The advantage of reusability enhancement through ALC approach. For the construction of a system *A*, the reusability gained in ALC process is higher than that in conventional processes.

$$ra(A) > rt(A) \quad (9)$$

V. REQUIREMENT DESCRIPTION LANGUAGE

RDL aids the ALC process by providing an approach to formalizing the requirements of common elements and the construction mechanism in an application domain.

Definition 6. Requirement Description Language (RDL). RDL is a formal language which is capable of describing the requirement of application systems within a domain. RDL is defined as a 2-tuple <*RE, CM*>, where

1. *RE* is a finite collection of elements, say, *Requirement Element*s. Each *re* ∈*RE* corresponds to a reusable entity constituting the requirements in a domain. A requirement element is given a specific name, and is characterized by its other two parts, that is, *P* and *M*. *P* is a collection of data properties and *M* is a collection of behaviors. Therefore *re*=<*Name, P, M*>.

2. *CM* is the *construction mechanism* for creating system instances based on the requirement elements. Generally, the requirement of a system can be described as an executing sequence of all the elements participating in it. Such a sequence is composed of flows of element behaviors as well as communications among elements, and *CR* is defined as the mechanism to establish a sequence of elements behaviors. By referring to the context-free grammar structures of conventional programming languages, the sequence can be described using three basic control structures: sequential executing, conditional branching, and looping.

The design process for an applicable RDL is based on extensive observation of in-existing systems. Several types of commonness in requirements are summarized, and the collection of reusable elements *RE* is derived from such commonness reflected in a large amount of particular instances. The common characters investigated involve at least the following types of commonness 1~8.

Commonness 1. The commonness of function specifications for user cases. Treating a user case as a black box, developers can understand its function only through the data items it manipulates and the functions it implemented. Using RDL as the description tool, all user cases' external requirements can be described as <*D, Actions*>, where *D* is the collection of input/output data items, and *Actions* is the executing sequence of element behaviors involved in the user case.

Commonness 2. The commonness of requirement parameterization. There exists in requirement an inherent character of parameterization. Taking user interface for example, it is common to see that a couple of dialogs can be parameterized from a common UI pattern [8], such as table-like pattern, navigation pattern, explore-like pattern, etc. By customizing appropriate attribute values of the pattern, a particular dialog instance can be created. For the elements in RDL, the data property collection *P* provides the mechanism of customization. The common function of an element is conducted by property values to perform distinct behaviors according to diverse requirement instances.

Commonness 3. The commonness of information structures. There exist the same structure styles of information in requirements. Consider a dialog receiving the user's input of his birth place, where three cascaded pull-down menus jointly represent the input items of "country - state - city". Such cascaded structure can be copied, and the contents can be slightly modified to represent the input item of the user's organization, displayed as "company - department - team - workgroup". Only the contents displayed in each menu are different. Regarding this commonness, a UI element named "cascaded input" is derived from these two instances. This element gains a larger granularity by grouping multiple pull-down menus, and provides properties which can be customized to adjust the levels and contents for a dedicated instance.

Commonness 4. The structural commonness of behavior sequences. Common structures exist in the executing flows of different requirement instances. Considering the UI validating task generally used to check the validation of users' input, a typical logic structure is: "At the moment of [a dialog's submission], check [each control] to see whether its content is validate; if there exists a control whose [value] doesn't matching a predefined [condition], cancel the submission and display some [error message] at [somewhere]". Regarding this commonness in diverse instances, the UI validating task can be derived as a common structure plus some attributes for customizing the structure. This structure is assigned as a standard logic all UI elements.

Commonness 5. The structural commonness of elements assembling. There exist common structures of the ways low-level elements are grouped into larger elements. For example, a typical UI structure of "pull-down menu + textbox" is frequently used for receiving approval results, where the pull-down menu provides several options for choice, and the textbox is used to fill in additional comments. This grouping structure is commonly seen in other requirements such as document review, application management, etc, therefore a UI element named "approval input" can be derived, which groups several small items to provide a pattern of user interface.

Commonness 6. The commonness of moments at which specific requirements are handled. In different requirement instances, some kinds of functions are usually performed at the same moments and by the same elements. For the example of UI validating task discussed above, it is usually handled at the moment of dialog submission; moreover, in dialog initialization, it is always the appropriate moment for loading initial data and preparing for display. Considering such commonness, specific functions are assigned as standard logic to be executed by an element at some moment.

Commonness 7. The commonness of object relationships. A series of common patterns can be derived to represent different kinds of relationships among requirement elements. The approach utilized in design patterns for object-oriented programming [4] gives a strong hint to the design of relationship patterns among business objects. For example, "Master-details" relationship extensively exists in business objects like product orders, which consists of a master record and several detailed records. Each detailed record represents a

product attached to the order. Such a pattern can be derived as an element representing the relationship of different requirement elements, which can be easily customized to represent business objects at a larger granularity.

Commonness 8. Mixed types of different commonness. Some requirements may have several types of above-mentioned commonness at the same time. Consider the dialogs displaying product orders in different systems. Although the attributes of orders can be different due to diverse business rules, all of them comply with a common "Master-detailed" relationship pattern. Meanwhile, the dialogs are usually designed according to an approximately the same layout style, that is, an area of controls displaying the master record plus a table displaying multiple detailed records. By summarizing such mixed types of commonness, requirement elements can be derived to take on larger granularity of reusable requirements.

According to above commonness, the common constitutions of system requirements are summarized, and a collection of requirement elements *RE* is derived. *RE* is composed of elements covering common features among domain systems. Considering the different abstract levels of elements, *RE* can be basically divided into two parts. The first part consists of fundamental elements which are atomic and rest in the lowest level of requirement; the second part consists of higher-level elements that are constructed by assembling the atomic elements in the first part through various types of requirement patterns.

A. Fundamental Requirement Elements

The collection of fundamental elements summarizes the basic constitutions at the lowest level of system requirements. In this level, each element is atomic and cannot be divided into other lower-level elements. As application systems can generally be separated into three distinct aspects, e.g. user interface, computational logic and external data access, the collection of fundamental elements can also be divided into three kinds.

1. *UI elements*: this kind of elements defines common constitutions which represents the interaction logic between users and systems. Each element acts as a displaying item or an interaction item, which has a larger granularity than conventional graphical controls, as shown in the previous examples of "cascaded input" and "approval input".

2. *Computational elements*: defining the common elements representing computation and calculation logic in system requirements. A series of large-granularity elements aiming at business data types along with corresponding operating rules are summarized in this collection, including multiple-attribute list, relational record set, structured documents, etc. Furthermore, a series of business-oriented computational instructions are equipped, including relational calculus, finance/statistics algorithms, traversing/iteration, etc.

3. *Data access elements*: defining the constitutions supporting data access logic occurring between systems and data sources. For generality, a series of data access primitives are defined. The requirements of access to several types of standard data sources can be described, such as relational databases and structured documents.

B. Requirement Patterns

Requirement patterns are a method to further promote the advantages of RDL. The patterns are derived based on the inherent commonness of different constructing styles in system instances. Such commonness refers to the previous Commonness 3~8. Each pattern characterizes a dedicated cooperating style of multiple fundamental elements. According to the three aspects of systems, three kinds of requirement patterns are derived, e.g., UI patterns, computational patterns, and data access patterns.

By grouping the fundamental elements, the scale of a pattern is naturally larger than that of each element it contains. As the functional scale is enlarged, the effort needed for reuse decreases. Therefore, patterns generally gain a higher degree of reusability than that of the fundamental elements.

It is notable that the collection of patterns has no absolute boundary. Due to the opening nature of requirements, higher level of patterns can be constructed at the basis of existing lower-level patterns. Therefore, new patterns can be appended to RDL in an incremental style, providing a method to hieratically expand RDL as needed. Plus the fundamental elements at the lowest level, the collection of all requirement elements are organized in a layering architecture. An architecture prototype and actual description instances can be found in Section 8 of this paper.

VI. System Assembling

System assembling mechanism is supported in ALC process through composition of multiple user cases. As the user cases can each be implemented by a large-granularity component, say, application-level component, a whole system can be constructed by assembling the group of ALC components implementing the corresponding user cases.

The user cases in a system may not be absolutely independent with each other. Instead, there may be couplings and interactions among some user cases; a higher-level user case may be composed by a series of lower-level user cases, where the latter provides functional services being requested by the former. This kind of relationships forms an aspect of application requirement at the level of system assembling, and is necessary to be supported by ALC.

Several current-existing methods for assembling of components can be referred to, such as framework [6] and connector [9]. The methods based on frameworks require additional assembling platforms aside from the component repositories; the methods based on connectors rely on specific representation elements to express the composition relationships and coupling details. They can all be classified into the category of heavy-weight assembling methods. In ALC process, a distinct assembling mechanism is utilized in a relatively lighter-weight style. This mechanism is aided by only one

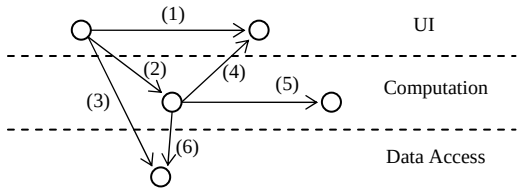


Figure 3. Component Calling Types

description primitive in RDL, which describes the calling logic between each pair of two components.

Definition 7. Component Calling Primitive (CCP). *CCP* is a basic primitive provided in RDL to describe the assembling relationships of user cases within a system. In the *DESC* which describes the requirement of a user case, a calling primitive may appear in the executing sequence as the form of

$$CCP = <Invoked\_ALOC, DM>$$

where *Invoked_ALOC* indicates a component to be called by the current-running component, and *DM* describes the data mapping relationship between the caller and the invoked component. DM is a series of 2-typle <*Caller_Data*, *Invoker_Data*>.

Generally, the executing semantic of *CCP* is as follows: when a *CCP* is parsed and executed by the current-running component, it first launches the component indicated by *Invoked_ALOC* into a ready-to-run state; second, the caller prepares initial input data for the invoked component according to *DM*, delivers the data to it, and triggers its running; next, the caller suspends itself and waits for the invoked component's completion of running; then, when the invoked component ends up, the caller takes over its output data; the data are finally transformed back to the caller as the service result of one request.

Such process provides a unified service-requesting mechanism which is business-independent. In this way, all kinds of couplings and relationships among user cases can theoretically be described with *CCP*. In RDL, *CCP* is designed as a shared instruction which is supported by all the requirement elements. Therefore, all user cases implemented by ALOCs gain the ability of assembling higher-level compound modules.

Because there are different types of requirement aspects in systems, the semantic of calling logic is slightly different according to the types of the caller and the invoked components. There are totally six supported calling types, shown in Fig. 3. The applicable occasions and semantic rules for each type are as follows.

Calling Type 1. UI → UI. Requesting lower-level UI service from a higher-level UI component. In this type, a portion of requirements in a business-oriented UI component can be fulfilled by invoking another UI component. The UI of the invoked component is entirely embedded the caller's UI. In this way, the invoked component's UI is reused as a whole, at the granularity of an user case. This mechanism enhances the reusability degree of UI resources which cannot be achieved by conventional small-grained graphical controls such as "textbox" and "button".

Calling Type 2. UI → Computation. Requesting service of computation logic from a UI component. In application systems, it is frequently required to execute computational operations inside user interfaces. As many computational operations are self-contained and have the potential of being reused in different occasions, this calling type enables components responsible for computational logic to be called by UI components, to fulfill the requirements as requested.

Calling Type 3. UI → Data Access. Providing data access services for UI components. By invoking data access components, data loading and persisting logic can be reused inside user interfaces.

Calling Type 4. Computation → UI. Calling UI components during computational logic, to provide necessary interaction interfaces as needed.

Calling Type 5. Computation → Computation. In a computation component, lower-level computation components can be called. This calling type can be used in a nesting style, which enables components to encapsulate computation requirements at different abstract levels.

Calling Type 6. Computation → Data Access. Loading and saving business data in computation operations whenever needed.

The above six calling styles jointly provide the ability to describe the assembling requirements for systems. By indicating the calling activities between each two components, the relationships among all user cases can be described in a bottom-up way. All the user cases can be assembled increasingly, and the whole system can eventually be constructed.

VII. ALC Structure Model

Application-level components have a common structure with which to realize their tasks. The basic task of ALC is declared in previous sections as parsing and implementing the requirement described in *DESC*, by executing the behaviors of requirement elements involved. All components implemented in the prototype comply with a structure model represented in Fig. 4. As discussed in Section 3, an ALOC consists of two parts which are separately defined as

$$ALC = <D, A, C, C_m, If>$$
$$DESC = <D', A', Actions>$$

1. The data properties of a requirement element is defined as two parts, e.g., *D* and *A*. *D* is the collection of data, and *A* is the collection of attributes. For each data property of a requirement element, if its value can be determined at the design time, it should be put into *A*; otherwise, if its value can only be determined at run time, it should be put into *D*. In accordance with this rule, the values of all the members in *A* should be tuned by developers at describing stage, according to the known requirement of the user case; whereas the members of *D* are generally treated as input/output data whose values are passed through from their calling components at runtime.

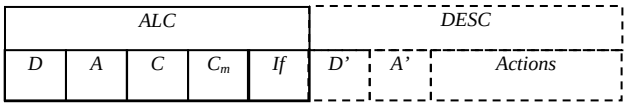


Figure 4. ALC Structure Model

2. The behaviors of a requirement element are defined as *C*. Each member in *C* is implemented an executable routine, which performs the equivalent software activity of a corresponding behavior.

3. C_m is a collection of executable routines which implements the standard running mechanism of the structure model. Four kinds of tasks are carried out by C_m: booting and establishing runtime environments at startup moment; responding to interface requests during runtime; parsing and executing *DESC* to implement the desired requirement; executing component calling primitives appearing in *DESC*.

4. *If* is the definition table of interface commands. Each command defines a kind of communication service which is supplied by the component. In practice, a minimum collection of *If* is shared by all the components, providing interface commands to supply declaration information for the developers. By invoking such commands, developers can inquire the contents of *D*, *A* and *DESC* from the component.

5. *DESC* is the essential part which implements description mechanism. The requirement of a user case is described by three parts. First, *D'* is used for describing the collection of data which is not publicly defined in *D* but is manipulated by a dedicated user case. Second, *A'* contains the values of all the members in *A* indicated by developers, according to the requirement of the user case. Finally, *Actions* describes the executing behavior of the user case by defining a sequence of executable routines selected from C_m.

The above constitutions each take different responsibilities in different development stages: the mechanism provided by C_m is shared by all the components based on the structure; *D*, *A*, *C* and *If* are specialized in each ALC, implementing the reusable function of a requirement element; *DESC* provides the description mechanism for implementing a dedicated user case, by providing requirement description information which is parsed by ALC. They jointly accomplish the task of creating business-oriented components by reusing existing ALCs.

The assembling mechanism discussed in Section 6 is supported by a portion of routines in C_m, that is, the routines executing *CCP*. As the semantic of *CCP* is basically defined from the viewpoint of running behavior, a collection of routines is adequate for completing the component invoking and data exchanging tasks. In this way, no additional constitution is needed for the assembling mechanism. This verifies the simplicity and light-weight nature of *CCP* method.

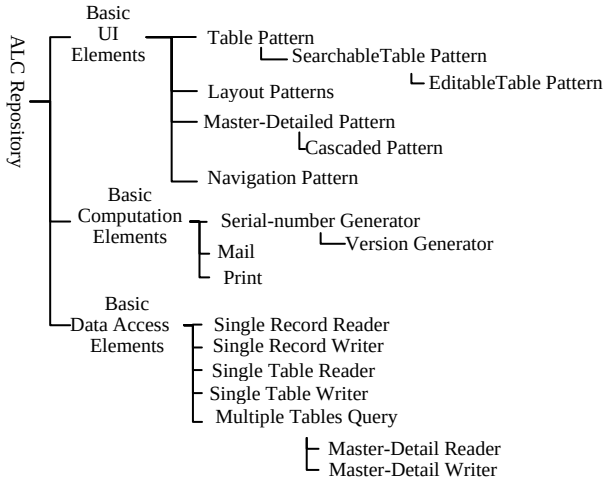


Figure 5. Prototype of an ALC Repository

VIII. Experiments and Evaluation

A. The Prototype of ALC Repository

To evaluate the practicability of ALC approach, an ALC repository prototype is established, shown in Fig.5. The prototype is reused in several system development processes, where measurements are carried out simultaneously.

Although the proposed repository is oriented to the domain of information application systems, efforts are currently being made to extend the reuse scope of ALC approach to more diverse domains, including operating systems, embedded systems, distributed middleware systems, etc. Preliminary evidences have been obtained, which reveal the promising effectiveness of utilizing ALC approach in various domains.

By observing the commonness of a large scope of practical system instances, a finite collection of basic requirement elements are summarized, and a series of patterns are derived subsequently.

B. System Construction

A medium-scale enterprise management system is selected to be built with the repository. The system consists of 85 user cases. Developers implement each user case by reusing an appropriate ALC and describing the desired function requirement with RDL.

Two user cases implemented by reusing the same ALC are shown in Fig. 6. The default interface of "Master-Detailed Pattern" is shown in Fig. 6(a). This pattern summarizes the commonness of "1:N" object relationship in systems, and provides a frequently seen UI framework supporting the necessary operations such as displaying/editing the master object, appending a new detailed object, selecting and deleting detailed objects, etc. By customizing the attributes of the pattern, more concrete business-oriented components can be created. The values of the attributes provide differentiated information for the two user cases. As shown in Fig. 6(b) and Fig. 6(c), the interfaces of two user cases, e.g., "document submitting" and "order entering" are all

created by reusing and customizing the same "Master-Detailed Pattern" component.

Revealed by the above example, at least two advantages are embodied by ALC approach:

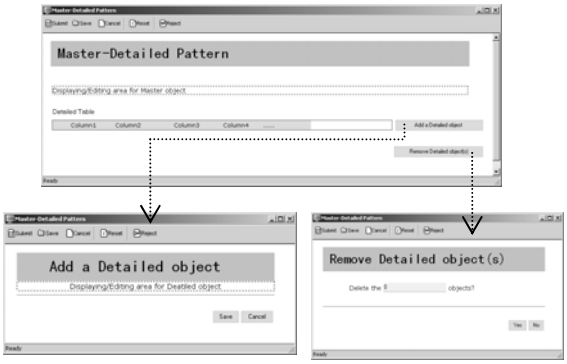
1. Only one component is needed to create a user case, which means the enlarged scale of granularity in ALC components. Therefore, a majority part of the user case's requirement has been implemented by a component, resulting in the sharp decrease of the description effort. It can be seen from the description content shown in Table 1 that, only 64 text lines are needed to create a user case, which can easily be done in several minutes.

2. Not only the effort for creating user cases decreases, the effort of modifying existing components to deal with requirement changes are also reduced. The effort for maintaining dozens lines of description content is definitely lower than the effort for maintaining complicated programming codes which are needed in conventional reuse processes.

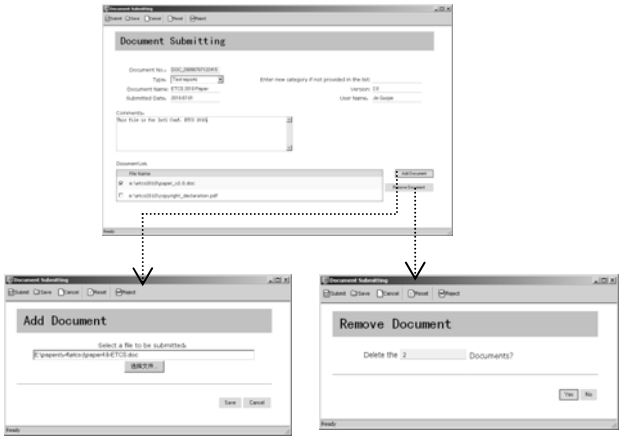
Table 1. Description Instance of "Document Submitting" User Case

```
/* System: Enterprise Management System
  * User case: Document submitting
  * Base component: Master-Detailed */

.args {
  Master_Title=Order Submitting
  Detail_Table_Title=Documents:
  Add_Detail_Title=Add Document
  Del_Detail_Title=Remove Document

  Master_Form={
   <Table rows="4" cols="4" align="R,L,R,L">
    (0,0)Document No.:
    (0,1)<Input id="DocumentNo" readonly>
    (1,0)Category:
    (1,1){CascadedInput "Category", ITEMS("Design specifications,
Meeting notes, Test reports, Applications, References")}
    (1,2)Enter new category if not provided in the list:
    (1,3)<Input id="NewCategory">
    (2,0)Name:
    (2,1)<Input id="Name">
    (2,2)Version:
    (2,3)<Input id=" Version" text="1.0">
    (3,0)Date:
    (3,1){DateInput "SubmissionDate"}
   </Comp:Table>

   Comments:<p>
   <Input id="Comments" rows="6" cols="70">
  }

  DetailTableColumns=<File Name>
  Item_Add_Form={
   Select File: {File "FileName" SIZE(80)}
  }
```

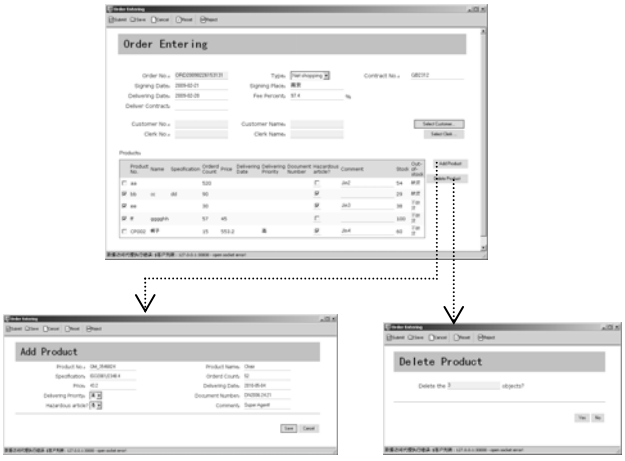
```
  FormPreprocessor={
     SubmissionDate := $FULL_DATE
     DocumentNo := CALL_COMP('SerialGenerator',
'DOC_$DATETIME')
     NewVersion := CALL_COMP('VersionGenerator',
$OriginVersion)
   }
  FormPostprocessor={
     Category := $NewCategory IF strlen($NewCategory) > 0
     Documents := PACK_FILES($DetailTable)
     CALL_COMP('IO_SR_Writer', 'INSERT files@db,
<$DocumentNo, $NewVersion, $Category, $SubmissionDate,
$Documents>')
   }

  FormChecks={
     strlen("$Name") == 0 ? Alert("Document name must not be
empty")
     $Version <= $OriginVersion ? Alert("New version must be
larger than the initial version")
     db_num_rows("$DetailTable") == 0 ? Alert("At least one file
should be choosn")
   }
}.args

.data {
  OriginVersion, String, I
  NewVersion, String, O

  DocumentNo, String, O
  Category, String, O
  NewCategory, String, L
  Name, String, L
  SubmissionDate, String, O
  Comments, String, O

  FileName, File, L

  Documents, Files, O
}.data
```



(a) Default UI of Master-Detailed Pattern



(b) UI of "Document Submitting" User Case



(c) UI of "Order Entering" User Case

Figure 6. Reuse Examples of "Master-Detailed" Component

C. Reusability Measurements

Four kinds of measurements are carried out along with the development cycle.

1. *Practicability of the ALC repository*. As families of commonness in a domain are summarized and implemented by the ALC repository, the development task of the enterprise management system is successfully accomplished by reusing the repository. Following the same way, future systems in the same domain can promisingly be built at a low cost. The diversity nature of requirements will never be an unbreakable obstacle for reuse.

2. *Entity Reusability*. This kind of reusability reveals the degree of reuse benefits obtained in system building stage. When creating a user case, the reusability of the component is measured according to Eq. (6) in Section 3. In practice, the efforts of E_c and $E_{description}$ are measured respectively according to the lines of *ALC* source code and the lines of description instance *DESC*, so the entity reusability is measured as

$$er(U)=\frac{LOC(ALC)}{LOC(ALC)+LOC(DESC)}*100\% \qquad (10)$$

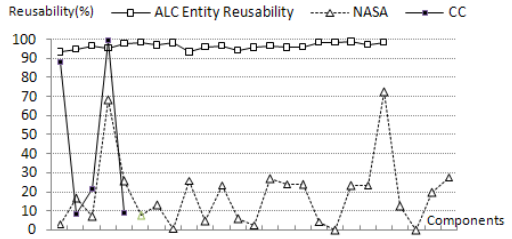
3. *Alternation Reusability*. This kind of reusability reveals the degree of reuse benefits obtained during system maintaining stage. When modifying a user case to fit the changed requirements, as there is no need to modify the ALC, the reuse effort lies only in the modification scale of *DESC*, so the alternation reusability is measured as

$$ar(U)=\frac{LOC(ALC)+LOC(DESC_{unchanged})}{LOC(ALC)+LOC(DESC_{After\_Alternation})}*100\% \quad (11)$$

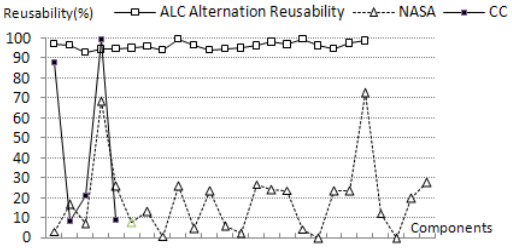
These two kinds of reusability are measured at each time of reuse, and the degrees revealed are shown in Fig. 7. The entity reusability of ALC reaches a high degree of 91.4~93.8%, while the alternation reusability reaches 92.5~95.7%. Compared with the average reusability of 17~32% from NASA [2] (or 81.9% at the maximum level), our method effectively gains a higher degree.

Another feature can be noticed in Fig. 7 that, the reuse degrees of different types of components are approximating balanced. This reflects the enhancement of reusability for all the components composing different aspects of requirements. This is once a problem not resolved in existing methods, as revealed by the data CC from Ref. [3].

4. *Cross-domains system development*. Aside from the enterprise management system, we put the repository into development for more domains, including ERP, OA, etc. Because such domains are all sub-domains of information systems, the repository can straightforwardly be reused in those domains. The reusability is measured at the peer-level with Fig. 7.



(a) Entity Reusability



(b) Alternation Reusability

Figure 7. Measurements of ALC Reusability

IX. Comparisons with Related Methods

Software reuse has long been a research focus in software engineering. Reference [4], [5] and [6] each proposes a kind of reuse process along with a corresponding component model. Although these models are extensively used in long-term development practice, the reusability of components is limited at a low level. Other literatures such as [10], [11] and [12] all emphasize different opinions on this phenomenon, but the problem remains.

With ALC approach proposed in this paper, a new kind of reuse process is established. Benefited from the enlarged granularity and the description ability, the degree of reusability obtained by ALC approach is enhanced to a higher level. Overall, developers can take two advantages from the new approach:

1. *The granularity of ALC components is larger than that of conventional components*. As the abstract level of both the basic requirement elements and the requirement patterns is upgraded to business-oriented level, the functional scale of such constitutions is larger than a low-level component. As a result, a user case can be implemented by only one ALC component. The effort needed for reusing an ALC component is decreased.

2. *ALC approach provides higher flexibility to deal with requirement diversity and variability*. An ALC component repository provides a completed collection of reusable elements for an application domain. Therefore,

an unbounded collection of diverse systems can be built by reusing such repository. Particularly, unforeseen system requirements can also be supported without modifying the repository. Once the requirement changes, the alternation effort for ALC approach is lower than conventional methods. Related methods concerning requirement describing language can be found in Ref. [13], [14] and [15].

The reusability is enhanced because of a combination of above factors. The ALC approach has the advantage of enlarging the reusable granularity while at the same lowering reuse effort. By incorporating ALC approach in development practice, systems can be built in a more efficient and lower-cost way.

X. Conclusion

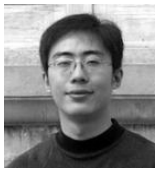
A new kind of reuse process named application-level component approach is proposed and proved to gain higher reusability than existing methods. The process is characterized by the new reuse style of "selection and modification". Compared with current methods, ALC provides the ability to cope with diverse and variable requirements on its own initiative, thus making it feasible to provide a repository which can be reused for implementing future unforeseen requirements. The detailed process of the approach is discussed, including the design of a requirement description language, the summarization of requirement patterns, the assembling mechanism, as well as the standard structure model. Measurements reveal the enhanced reusability at 92.5~95.7%, which verify the effectiveness of the approach.

Acknowledgment

This work is supported by the Chinese State Key Laboratory Exploration Fund under Grant No. SKLSDE-2009ZX-11.

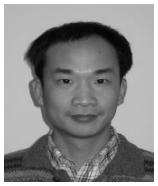
References

[1] Parastoo Mohagheghi. An Empirical Investigation of Software Reuse Benefits in a Large Telecom Product, ACM Trans of Software Engineering and Methodology, 2008.3
[2] Richard W. Selby. Enabling Reuse-based Software Development of Large-scale Systems, IEEE Trans of Software Engineering, 2005.6
[3] Huang Liuqing, Wang Manhong. New Software Engineering: A Component-Oriented Approach, Tsinghua University Press, ISBN 7-302-12925-8, 2006.5
[4] Alan W. Brown, Large Scale Component-Based Software Development, Prentice Hall, 2001
[5] Kang K, Cohen S, el. FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. Annals of Software Engineering, 1998.5
[6] Elsenecker, K. Czarnecki, el. Generative Programming: Methods, Techniques, and Applications, GPCE '03
[7] Lin Liu, Fei Peng, el. Understanding Chinese Characteristics of Requirements Engineering, Intl Conf of Requirement Engineering 2009
[8] Trudy Levine. Reusable software components, ACM SIGAda Ada Letters, ACM Press, NY USA, 2005.9
[9] S. Sentilles, el. A Component Model for Control-Intensive Distributed Embedded Systems, CBSE '08
[10] Ciliane Redolfi, el. A Reference Model for Reusable Components Description. 38th Hawaii Intl. Conf. System Sciences, 2005
[11] Philip T Cox. A Formal Model for Component-Based Software. HCC 2001
[12] Lin Liu, Fei Peng, Tomas Burda, el. Understanding Chinese Characteristics of Requirements Engineering. ICRE '09
[13] J.S. Lee, H.S. Chae. Domain-specific language approach to modelling UI architecture of moblie telephony systems. IEEE Proc. Software(online), 2006.6
[14] Pedro J. Molin. Just-UI: Using Patterns as concepts for UI specification and code generation. CHI'2003 Workshop
[15] J. S. Cuadrado, J. G. Molina. A Model-Based Approach to Families of Embedded Domain-Specific Languages. IEEE Trans of Software Engineering, 2009.6



Jin Guojie is now a PhD cadicate of computer science in School of Computer Science, Beihang University, Beijing, China.

His research interests includes component technology, software reuse, and workfow systems.



Zhao Qiyang received his PhD degree in computer science from Beihang University, Beijing, China, in 2008.

He is now an associate professor with School of Computer Science, Beihang University.

His main research interests include software engineering, computer vision, and digital watermarking.

Yin Baolin received his PhD degree from Edinburgh University, UK, in 1984.

He is now a professor and PhD supervisor with School of Computer Science, Beihang University.

His main research interests include distributed systems, software reuse, and workflow systems.