11-1-2001

Component-Based Software Development

Luiz Fernando Capretz
University of Western Ontario, lcapretz@uwo.ca

Miriam Capretz
University of Western Ontario, mcapretz@uwo.ca

Dahai Li
dli@gmail.com

27th IEEE Conference of the Industrial Electronics Society (IECON'01), Denver, November 2001

Component-Based Software Development

Luiz Fernando Capretz
Miriam A. M. Capretz
Dahai Li
Department of Electrical & Computer Engineering
University of Western Ontario
1201 Western Road
London, N6G 1H1, CANADA
{lcapretz,mcapretz,dli7}@uwo.ca

***Abstract* – Component-based software development (CBSD) strives to achieve a set of pre-built, standardized software components available to fit a specific architectural style for some application domain; the application is then assembled using these components. Component-based software reusability will be at the forefront of software development technology in the next few years. This paper describes a software life cycle that supports component-based development under an object-oriented framework. Development time versus software life cycle phases, which is an important assessment of the component-based development model put forward, is also mentioned.**

I. INTRODUCTION

Components are the *Lego* blocks of software engineering. Component-based software development has become increasingly important in the software industry, with some observers predicting that in the near future many software systems will be produced reusing components. Software manufactures applying component-based software development not only benefit from reduced development time and costs through the systematic reuse of in-house and off-the-shelf components, but also have a powerful technique for handling complexity. There are COM+ [1] from Microsoft, Enterprise JavaBeans [2] from SUN, and Component Broker from IBM [3], CORBA [4] from Object Management Group, among other projects that offer off-the-shelf components for software construction.

Several software life cycle models have been proposed. It is appropriate to examine different software development models in general and point out their strengths and weaknesses before an alternative one is put forward. Even though the waterfall model [5] has long been used by software engineers; it takes no account of bottom-up development and prototyping.

The spiral model [6] has been proposed mainly to speed up software development through prototyping, but without a clear and explicit goal, this process can degenerate into uncontrollable hacking. The fountain model [7] supports incremental and iterative software development, which takes place during the production of object-oriented software. However, one of the main shortcomings of such models is that none of them explicitly encourages reusability along their phases. Therefore, a component-based software development model is still very much in demand.

II. A COMPONENT-BASED DEVELOPMENT MODEL

The creation of software is characterized by change and instability, and therefore any diagrammatic representation of a component-based development model should consider overlapping and iteration between its phases, a consensus may be drawn on the phases pertinent to such a model. Although the main phases may overlap each other and iteration is also possible, the planned phases are: domain engineering, system analysis, design and implementation.

Fig. 1 displays a pictorial representation of how these phases proceed iteratively over time; and how reuse of components from a reusable library is taken into consideration within the software development model. Reusability within this model is smoother and more effective than within traditional models because it integrates at its core the concern for reuse and its mechanisms. Although maintenance accounts for the majority of software costs, it is not included in Fig. 1, because it can be viewed as an operational phase, in which bugs are corrected and extra requirements met, but that succeeds software development.

A feature of this software development model is the emphasis on reusability during software creation, and the production of reusable components meant to be useful in future projects. This is naturally supported by the object-oriented paradigm due to inheritance and encapsulation. Reusability also implies the use of composition techniques during software development. This is achieved by initially selecting reusable components and aggregating them, or by refining the software to a point where it is possible to pick out components from reusable libraries.

A. Domain Engineering

Domain engineering is about finding commonality among software systems in order to identify components that can be applied to a family of systems rather than one single system. It deals with the analysis and modelling of a given application domain, which will provide scope for future software systems. Thus, domain engineering is an activity that should be carried out at the beginning of software specification if reuse is to be considered. As domain engineering can yield an initial taxonomy reflecting the main conceptual entities within an application domain, essential properties of that domain are

captured and initial candidates for reusable components emerge.

The domain engineering process starts out with the domain to be analyzed. As its primary source of information, domain engineering relies on existing applications and experts on the respective domain. On the basis of objects, operations and relationships that have been identified as reoccurring across the domain and thus being amenable for reuse, the process yields a domain model. This acts as a guide to identify and categorize potentially reusable components that will be subsequently implemented. Further inspection of the domain can help in building a domain vocabulary, which increases the expressiveness for describing the domain from a software engineering point of view.

To illustrate, a process control system for a chemical plant is concerned with vessels, pipes and valves of that plant, as well as the flow of liquid and gases, the temperature and pressure at various points in that plant. A payroll system is concerned with employees, the pay they earn, the tax they owe, and the holidays they are entitled to. These real-world entities and interrelationships are likely to become part of the vocabulary for those application domains.

User needs, software requirements, functionality, objectives and constraints of the system are very much of interest during the system analysis and domain engineering phases. Thus, it is important to understand the real-world application, and an abstract model of that application should be depicted. Therefore, the boundary between system analysis and domain engineering may at times seems fuzzy because identifying key abstractions in the application domain may be viewed as part of domain engineering or system analysis. Nevertheless, at this level, domain engineering is also concerned with the identification of potentially reusable components.

B. System Analysis

This phase involves high-level analysis of the application with the purpose of understanding its essential features. The system analysis phase demands the system analyst to:

- study the application and its constraints;
- understand the requirements expected to be satisfied by the software system;
- create an abstract model of the application in which these requirements are met.

This phase may conduce to the identification of the major parts of the application, so that the system can be divided into large components based on the functionality that should be offered. A glimpse of the preliminary components that model the application can come up as well.

At this stage, the services delivered by a software system helps figure out its subsystems and major components. However, as compared to functional decomposition, this phase is neither concerned with the details of functions in terms of algorithms, nor which functions can be refined into other sub-functions, but it worries over mapping the application in terms of components. The result of this phase is an abstract model of the application, which may be graphical or textual, using a formal or informal method, as the systems analyst wishes.

C. Design

Design is an exploratory process. The designer looks for components trying out a variety of schemes in order to discover the most natural and reasonable way to build the software application. There has been a tendency to present software design in such a manner that it looks easy to do. Nevertheless, in the design of large and complex software, identification of key components is likely to take some time.

During the design phase the primary concern is to build a design model comprising both the static and dynamic concepts, which fulfils the overall software functionality. The construction of the design model involves identifying relevant components, and producing the design model.

When designers face an application, they should not ask "How do I work out a solution to this problem?" Instead, they should ask, "Where are components that I can directly or indirectly reuse to solve this problem?" At this point, they should be able to examine reusable libraries to select components that closely match the ones necessary to build the software.

As more components are identified along the design, re-evaluation of the complete set of components is required. Repetitions are not unusual, since a good design usually takes several iterations. The number of reiterations also depends on the designer's insight, experience and knowledge about the application domain. A bottom-up strategy should be considered if the software engineer does not have a good perception of the application domain.

Some components picked out during the design phase should undergo further refinements (e.g. treatment of exceptional conditions) until they become generic and robust enough to be placed in a reusable library. This surely adds an overhead to software construction, which is more than compensated for by the long term savings when such components are reused in future projects.

D. Implementation

The implementation phase is characterized by the translation of a design model into correct programs, so it is assumed that testing and debugging are part of the implementation phase. The design model comprises static concepts and dynamic behaviour represented by the output of the design phase.

In this phase the major tasks involve the implementation of identified components, along with the cooperation among them, in order to fulfil the required software functionality. The best idea is to isolate a component and decide whether a match can be reused, or if it has to be implemented from scratch.

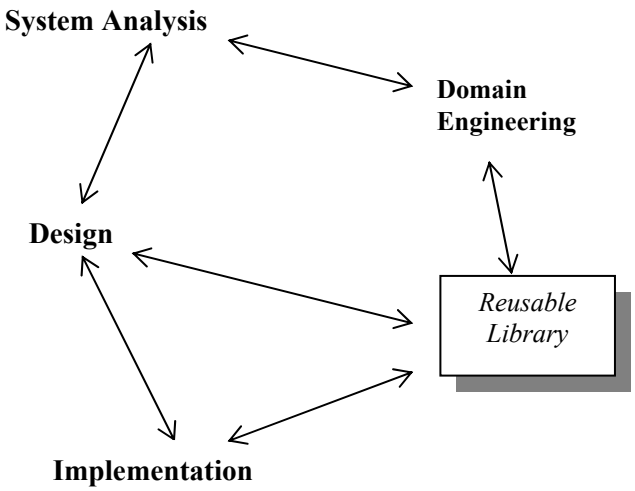

Fig. 1. A component-based software development model

III. SOFTWARE DEVELOPMENT ISSUES

The graphical features of a CASE environment [8] have been developed reusing an interactive framework following the component-based process presented. The experience of using the component-based software development model to develop large and complex software systems has firstly shown that it is very difficult to follow either a strictly top-down or bottom-up approach, it is necessary to switch over between them. This implies that it is helpful to clarify high-level functionality for the software along with the identification of some low-level components and study their interactions. As a result, when developing large software, it is important to synthesize ideas from both top-down and bottom-up directions.

One great advantage of the proposed model is the conceptual continuity across all phases of the software development. Not only do the software concepts remain the same from system analysis down through implementation, but they also stay uniform during the refinement of a design. Therefore, when that model is followed, the design phase is linked more closely to the system analysis and the implementation phases because software engineers have to deal with similar abstract concepts throughout software production.

Although it is difficult to draw distinct lines between two adjacent phases, it is worth indicating an approximate percentage of the amount of time likely to be spent on each phase for a complete development of a system. The numbers are: domain engineering (25%), system analysis (25%), design (40%) and implementation (10%). These statistics have been taken from the construction of a few small software systems. Despite the system analysis, design and implementation phases being deeply interrelated, it is clear that the design phase takes longer because most of the refinements are done during this phase.

Domain engineering is relevant to figure out potentially reusable components during software production. Consequently, the amount of time spent on this phase, naturally, must not be longer than that spent on other phases. If the perceived cost of finding a certain component is higher than the cost of creating a new component from scratch, then all hope for reuse is lost. For this reason, it is important to have at least minimal library tools that allow software engineers to select and manipulate components.

IV. RELATIONSHIP BETWEEN COMPONENTS

So far, most of the work that has been done in the reusability arena involves storing and recovering components from reusable libraries, but there are still many problems related to reusing such components. For instance, as a software system becomes mature, the reusable libraries may grow as domain-specific libraries and reusable components can be added over time. It does not take long for such libraries to expand to enormous proportions and often with multiple versions of a component, which makes it difficult for software engineers to look for components, which might meet their needs. Reusable libraries are usually large and their organisation makes it problematic to find potentially reusable components.

Additionally, one of the great difficulties in identifying a reusable component lies in the fact that there is discordance in terminology among different professionals, in that a component someone is looking for might be described in a library by unfamiliar or unexpected terminology.

Ideally, the potential re-user of software components must be able to find a connection between what is needed and what is available. Relationships between components could be used to facilitate the search for potentially reusable ones. For instance, *has-a* relationships could be described in composition diagrams, *is-a* relationships are presented in class hierarchy diagrams, *uses-a* relationships can be depicted from operations, and *is-part-of* relationships can relate a component to a particular context or framework. Such relationships can be seen as a classification scheme to provide a network of pre-defined links between components, thus introducing some semantic information and a vocabulary into a reusable library.

One way to express relationships between components of a reusable library involves organizing them through a set of pre-defined relations. Such relations allow components to be classified, and correlated to others that could also be reused. In addition, relations can be used to express a link between different components, facilitating the understanding of the components. Relations used to represent information between two reusable components can help solve the problem of discordance of terminology among professionals because the relations can establish some fixed semantic concepts between components.

Four different relations to link components and to express relationships among components include:

1. **Compose** (<component-1>, <list-of-components>): This relation represents <component-1> as a composition of components in a <list-of-components> (*has-a* relationship). Complex software system behaviour can be achieved with compositions that combine the simple behaviour of several types of objects.
2. **Inherit** (<component-1>, <component-2>): This relation indicates that <component-1> is a generalization of <component-2> or the other way round that <component-2> is a specialization of <component-1> (*is-a* relationship). This information can be found in any class hierarchy diagram.
3. **Use** (<component-1>, <list-of-components>): This relation indicates that <component-1> interacts with components in a <list-of-components> (*uses-a* relationship). It means that any operation of <component-1> uses any operations defined in any component in a <list-of-components>.
4. **Context** (<component-1>, <context-1>): This relation associates a <component-1> with a <context-1> defined by the software engineer (*is-part-of* relationship). The <context-1> can be a framework.

There are differences in the mechanisms used to achieve reusability when different kinds of reusable components are involved. The most basic software components are often reused by composition, which can be seen as a process of building a piece of software from elementary self-contained components. Nevertheless, reusability is naturally accomplished reusing classes through inheritance during object-oriented software development. In this case, it takes place by specialization and generalization of commonalities between classes.

If a newly implemented component does not exist in the reusable library, then a decision has to be made as to whether the new component should be classified as a reusable component, and to be validated and put in a reusable library. Not all classes identified early in the development process are implemented because some of them can be refined during the design phase or taken from a library of reusable components. It is better to reuse high-level components such as classes during design because they have fewer implementation details, which would limit their applicability.

V. CONCLUDING REMARKS

There is a need for tools to support the creation of domain-specific collections of reusable components, also known as *framework*, which is tuned specially for a particular application domain, i.e., an interface-building framework. A framework comprises a set of components that express a design for a family of related applications. Therefore, a framework will not be as generally useful outside the application domain because it contains domain-dependent components. However, it is sometimes beneficial to adapt the developing software so that it fits to an available framework, resulting in a tremendous gain in productivity.

The graphical features of a CASE environment have been developed reusing an interactive framework following the process described above. The results obtained from that implementation clearly show that the application of component-based technique can substantially increase software development productivity and reliability. There is an important lesson that should be learned from that work, that is, several independent reusable libraries are more effective for reuse than a single universal library of components. Therefore, rather than creating a single library as a centralized repository of components, a better strategy is the development of specific reusable libraries for certain application domains.

VI. REFERENCES

[1]Microsoft,*COM+*,http://www.microsoft.com/com/tech/complus.asp, (July 2001).
[2]SUN,*JavaBeans*,http://www.java.sun.com/products/ejb/index.html, (July 2001).
[3]IBM,*ComponentBroker*,http://www.software.ibm.com/ad/cb, (July, 2001).
[4]ObjectManagementGroup, *CORBA*, http://www.omg.org, (July 2001).
[5] W. W. Royce, "Managing the development of large software systems," In *Proceedings of IEEE 9th ICSE* (Monterey, CA), IEEE Press, 1987, pp. 328-338.
[6] B. W. Boehm, "A spiral model of software development and enhancement," *IEEE Computer*, vol. 21, no. 5, May 1988, pp. 61-72.
[7] B. Henderson-Sellers and J. M. Edwards, "The object-oriented systems life cycle," *Comm. of the ACM*, vol. 33, no. 9, Sep. 1990, pp. 142-159.
[8] L. F. Capretz, "A CASE of reusability," *JOOP,* vol. 11, no. 3, June 1998, pp. 32-37.