ENGINEERING & TECHNOLOGY

Khulna University Studies 3(2): 513-519

BENEFITS OF USING COMPONENT-BASED APPROACH OVER CONVENTIONAL ONE OF DEVELOPING LARGE-SCALE AND COMPLEX SOFTWARE SYSTEM

M.K. Islam* and M.S. Hossain

Computer Science and Engineering Discipline, Khulna University, Khulna-9208, Bangladesh

KUS-01/27-210801

Manuscript received: August 21, 2001; Accepted: April 01, 2002

Abstract: The idea of conventional software development has been changing from last few years because of rising demands of various software products within short period of time. Developing software from scratch is the main concepts of conventional software development considered time consuming and costly methods. Building software from existing software systems using reusable components is considered important in reducing development cost and time. In this paper, we have highlighted on both Component-Based and co conventional software development approach and found out their merits and demerits in the context of developing reliable and efficient large-scale and complex software products.

Keywords: LRUsim;Large-scale;Component-based Software Develoment;Object-oriented;LEAPs.

Introduction

Developing reliable and efficient large-scale software systems within minimum time and cost is the main concern of expert software professionals. Traditional software development approach, which starts from scratch, is unsuccessful to run with the current software needs. Software productivity is steadily rising over the past few decades. However, increasing the number of software professionals has not kept up with the rising demand for developing new ever more complex and large software systems and maintaining current software. Component-Based Software Development has raised the hope of developing more reliable and complex software systems within minimum cost and time. With Component-Based Software Development, software productions will become easier, more less expensive, more reliable, more efficient, less time to develop and less expensive to maintain.

Currently it is rare that software developed by one development group is reused by another, and integration options are limited by the predominant "vertically integrated" applications paradigm. A Component-Based Software Development process can help the software industry to use the reusable software components properly. In this paper, we have discussed in details both Component-Based Software Development and Conventional approach and found out their relative merits and demerits.

Component-Based Software Development Process

In Component-Based Software Development, the notion of building a system by writing code has been replaced with building a system by assembling and integrating existing software components. In contrast to traditional development, where system integration is often the tail end of an implementation effort, component integration is the centerpiece of the approach; thus, implementation has given way to integration as the focus of system construction. Because of this, integrality is a key consideration in the decision whether to acquire, reuse, or build the components. A typical Component-Based Software Development should contain the following steps :

- Identification
- Classification
- Storing in a repository
- Retrieval
- Adaptation
- Composition.

*Corresponding author: Tel: 88-041-721791,720171-3, Fax: 88-041-731244; E-mail: cseku@khulna.bangla.net

DOI:https://doi.org/10.53808/KUS.2001.3.2.0127-se

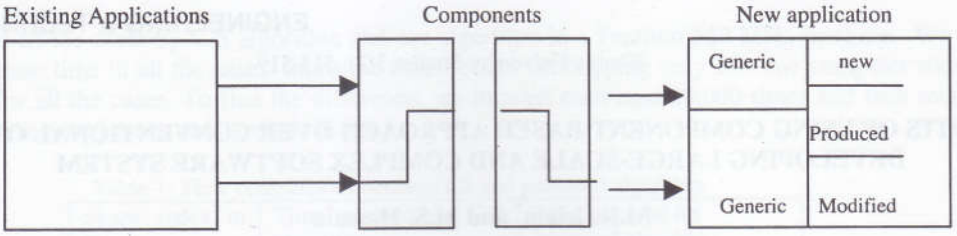

Fig. 1. Component-Based Development.

More detailing, they are processed like fig. 1 and using the following steps:

- Partition and extract components from the existing system (using generalization, restructuring and abstraction tools)
- Store the components in a repository
- Find out components that can be reused
- Modify the components to meet specific user requirements
- Build new components if not available in the repository
- Make the extracted components, the modified components and the new-built components the off-the-shelf components
- Supply the off-the-shelf components to users, so that they can make software that meet their specific requirements by simply assembling them

Comparison

Although object-oriented technologies have promoted software reuse, there is a big gap between the whole systems and classes. To fill the gap, many interesting ideas have emerged in object-oriented software reuse for last several years. They include software architecture design patterns and framework (Fayad and Schmidt, 1997). We have earlier mentioned on such reusable software elements.

Component-Based Software Development takes different approaches from the conventional software reuse in the following manner.

- **Plug and Play:** Component should be able to plug and play with other components and/or frameworks so that component can be composed at run-time without compilation.
- **Interface-centric:** Component should separate the interface from the implementation and hide the implementation details so that they can be composed without knowing their implementation.
- **Architecture-centric:** Components are designed on a pre-defined architecture so that they can interoperate with other components and/or frameworks.
- **Standardization:** Component interface should be standardized so that they can be manufactured by multiple vendors and widely reused across the corporations.
- **Distribution through Market:** Components can be acquired and improved though competition market, and provide incentives to the vendors.

The nature of Component-Based Software Development suggest that the model of component-based software development should be different from the conventional development model. Table 1 summarizes major characteristics of conventional software development and component-based software development, which are briefly discussed in the following sections.

Table 1. Comparison of Development Models.

.Characteristics	*Conventional*	*CBSD*
Architecture	Monolithic	Modular
Components	Implementation and White-Box	Interface and Black-Box
Process	Waterfall and Big-bang	Evolution and Concurrent
Methodology	Build from scratch	Composition
Organization	Monolithic	Specialized : Component Vendor, Broker and Integrator

Architecture

Component-Based Software Development emphasizes modular architecture so that we can partially develop a system and incrementally enhance the functions by adding and/or replacing components. To make such

design possible, we need a sound foundation of software systems, that is, software architecture. Most component-based systems assume underlying software architecture such as MFC (Microsoft Foundation Class) and CORBA. They are provided in the form of frameworks. Frameworks are workable reference to the underlying software architecture.

To be effective, framework can be hierarchical up from domain independent to domain specific. Examples include Andersen Consulting's Eagle project and IBM's San Francisco project (Lazar, 1998). With standardized and modular software architecture, the CBSD can avoid ad hoc and monolithic design.

Process

Component-Based Software Development makes software development and delivery be evolutional. Since some parts of a system can be acquired from the component vendors and/or be outsource to other organizations, some parts of software process can be done concurrently.

Architecture of Software Process

To make software reuse happen, software process should be reuse-oriented so that designers can reuse artifacts at different levels of abstraction along with software process. Figure 2 illustrates conventional waterfall process and an example of Component-Based Software Development process.
Component-Based Software Development process consists of two processes; component development and component integration. Since these two processes can be done by different organizations, these two processes can be concurrent. Unlike conventional process, Component-Based Software Development process need a new process for component acquisition.

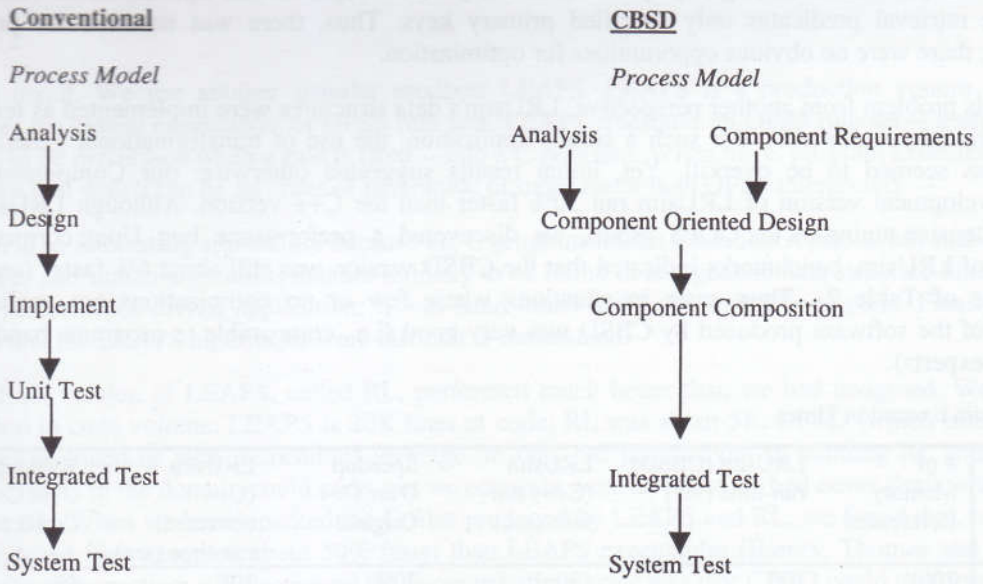

Fig. 2. Conventional Process and CBSD Process.

Methodology

Methodologies need to deal with both component development and component composition. Most of conventional methodologies such as object-oriented methodology assume development from scratch and have not provided much help for reuse-oriented development. Furthermore, plug and play software components separated interface from the implementation and provide interface Component-Based Software Development focuses on composition of components through their interface. Composition also requires to design collaborative behavior of multiple components. So, Component-Based Software Development methodologies need to help interface-centric and behavior-oriented design such as Catalysis (Souza and Wills, 1998) and connection-oriented programming.

Organization

The separation of component development and component integration created a new role of component broker. Component broker can sell and distribute software components. Since component development and component integration requires different expertise, it is natural to specialize the organizations into

component vendors and component integrators. This specialization will requires the mediators between two organizations, that is, component brokers. This organization structure can be called vendor-broker-integrator model (Ning, 1996). As the software component vendors have been growing, a software component market is emerging. Since software can be distributed over the Internet, web-based software component brokers have emerged (Aoyama et al., 1998).

Comparison Results

Result: 1

Here we take the comparison results of Component-Based Software Development versus conventional software reuse using LRUsim products. The Object-Oriented Programming Systems (OOPS) Group at the University of Texas is investigating issues of OS memory management. One of their projects is to analyze the locality of references in memory using real program traces. Their observation is that randomly generated memory references do not reflect the true usage of memory; the proper way to evaluate memory management schemes is to use real traces.

LRUsim is a hand-coded C++ tool that maintains the LRU ordering of pages using container data structures. LRUsim reads a trace file and maintains a page reference queue; it records the position of each page in the queue at reference time and promotes that page to the head of the queue. If the memory size is m pages, references to positions 1...m represent page hits; above m are misses (page faults). The idea naturally extends to memory hierarchies: level 1 has m pages, level 2 has n pages, etc. LRUsim is an interesting application for Component-Based Software Development for many reasons. First, performance is critical: LRUsim runtimes are measured in hours or days. Second, LRUsim's data structures are C++ templates. They are relatively simple structures (e.g., splay trees) that require no sophisticated optimizations. Not so for LRUsim: the retrieval predicates only specified primary keys. Thus, there was no need for predicate manipulation; there were no obvious opportunities for optimization.

Looking at this problem from another perspective, LRUsim's data structures were implemented as templates (i.e. compositional components). For such a simple application, the use of transformational generators in CBSD process seemed to be overkill. Yet, initial results suggested otherwise: our Component-Based Software Development version of LRUsim ran 30% faster than the C++ version. Although LRUsim had undergone extensive tuning by the OOPS group, we discovered a performance bug. Upon correcting the C++ version of LRUsim, benchmarks indicated that the CBSD version was still about 6% faster (see right-most columns of Table 2). Thus, even in situations where few or no optimizations are applied, the performance of the software produced by CBSD was very good (i.e., comparable to programs hand-coded and tuned by experts).

Table 2. LRUsim Execution Times.

Trace File	# of Memory References	LRUsim (CBSD) run-time (sec)	LRUsim (C++) run-time (sec)	Speedup Over C++ Original	LRUsim (C++ corrected) run-time (sec)	Speedup Over C++ Corrected
Tex	200K	100	130	30%	108	8%
Spice	200K	115	153	33%	124	8%
Ghostscript	107M	64K	84.2K	31%	66.3K	4%
Expresso	592M	378K	482K	27%	402K	6%

We learned important lessons from this experiment: synthesized algorithms have a higher probability of performing better (and for the same reasons are more reliable) than hand-coded algorithms. Remember, we are not using formal methods to generate software, nor are we synthesizing customized algorithms (Smith, 1990), but merely gluing pre-written algorithms together. So our approach to software synthesis is quite simple.

The reason for improved performance is that the authors of generator components focus their attention on coding the most efficient implementation of a set of algorithms for a highly constrained problem. (In the case of Component-Based Software Development, these are algorithms for a particular data structure). Consequently, much more effort is focussed on optimizing what would otherwise be called "minute" coding details.

Coding a data structure from scratch that corresponds to a composition of components typically requires far too many details to keep straight and to optimize. It is impractical and too time consuming for programmers to optimize code at the same level of detail. Programmers have finite time and energy, and attempt only a fraction of these optimizations. For this reason, the synthetic algorithms manufactured by CBSD will often outperform their hand-written counterparts.

Stated another way, components generate locally optimal code. Composing locally optimal code fragments doesn't guarantee global optimality. So it is always possible to handcraft code that will out perform that which is produced by a generator. But the issue is "At what cost?". Just as it is possible for humans to outperform optimizing compilers by hand-coding assembly statements, it is generally not considered practical; the quality of compiler output is good enough compared to the productivity advantages gained in programming in higher-level languages. So too it seems for using generators.

Components represent repositories for "best practice" approaches for building software: they can encapsulate coding tricks and proven-effective approaches for implementing their "feature" in an efficient way. Over time, additional effort is put into components so that the code that they produce improves even further, so that it is better (on average) than anything individuals or even groups of programmers can produce. Thus, composing locally optimal components produces very good software.

It is interesting to note that there were productivity improvements in building LRUsim with CBSD. The C++ version of LRUsim is 6600 lines; the Component-Based Software Development version is about 2500 lines. The largest CBSD function has 52 lines of code; CBSD expands this to 1300 lines. Once again, by raising the level of abstraction, the complexity of an application is substantially reduced. In our exit surveys with the OOPS group, they believe that the ability to compactly write complicated code is CBSD's major benefit. It enabled revisions and restructuring that they believed would be difficult or impossible to perform by hand.

Result: 2

In this result, We use another popular products LEAPS. LEAPS is a production system compiler that produces the fastest executables of OPS5 rule sets. LEAPS translates an OPS5 rule set -- a set of rules with actions to be performed when a rule is fired -- into a C program. When the C program executes, it fires these rules at a rate which can be an order of magnitude or more faster than OPS5 interpreters.

LEAPS is an interesting application because the C programs that it generates relies on unusual container data structures and search algorithms that are unlikely to be found in any generic data structure library. LEAPS is also a performance-driven application; it was hand-tuned and hand-coded by experts. Finally, it was well known that the LEAPS algorithms were difficult to understand.

Our CBSD version of LEAPS, called RL, performed much better than we had imagined. We had a 4-fold reduction in code volume: LEAPS is 20K lines of code; RL was about 5K. CBSD (which itself is 50K lines of code) provided us with tremendous leverage. In fact, our productivity in building RL was at the rate at which experts in the domain could code, yet we certainly were novices: we had never dealt with rule systems prior to RL. When we benchmarked the C files produced by LEAPS and RL, we found that, on average, the RL-produced files executed about 50% faster than LEAPS executables (Batory, Thomas and Sirkin, 1994). Upon closer inspection, we discovered that a contributing factor was that CBSD could perform optimizations automatically that were difficult, if not impractical, to do by hand. Finally, our CBSD specifications of the LEAPS algorithms (expressed in terms of cursors and containers) were very clean and easy to understand. It revealed the underlying elegance of LEAPS, but also suggested ways in which to improve its performance by swapping/adding CBSD components to the type equations that defined the implementations of LEAPS containers.

Figure 3 shows the performance results for a typical rule set, waltz. The Y-axis indicates run-time in seconds (logarithmic, base 10); the X-axis shows data set size. The top two curves show the performance of two versions of LEAPS: the highest (slowest) is the persistent version called DATEX. (DATEX containers resided in persistent storage that used the relational storage manager of Genesis(Batory, Barnett, Garza, Smith, Tsukuda, Twichell and Wise, 1988). LEAPS (which used transient containers) is the second highest curve. The performance of RL (for both transient and persistent versions) is the two lines below that of LEAPS. The persistent version of RL was achieved by unplugging the transient component and replacing it with a memory-mapped persistent component. (Because memory mapped I/O only results in a degradation of about 10%, there was little performance difference between transient and persistent versions of RL).

We mentioned earlier that because the CBSD specification of LEAPS revealed the source of much of the CPU overhead, we introduced new hashing components/data structures to the type equation that defined RL containers. By doing so, we were able to improve performance further (the bottom-most curves of Figure 3). Our persistent hashed-version of RL ran over 40 times faster than LEAPS! (Batory and Thomas.)

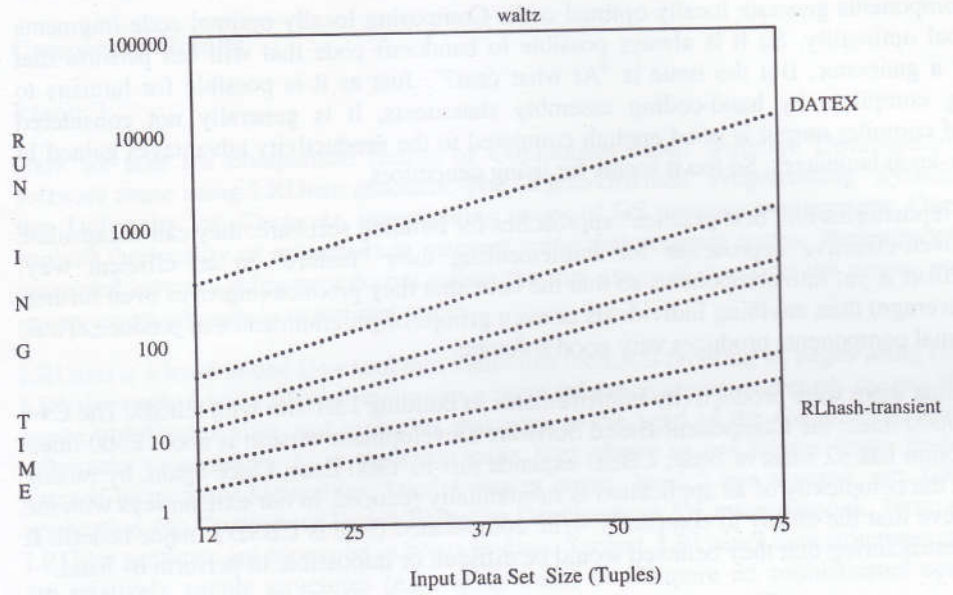

Fig. 3. Performance Comparison of RL with LEAPS running time

We learned an important lesson in software quality from our experiments: abstracting voluminous and complex implementation details promotes clean, efficient, and understandable designs. Moreover, organizing fewer details often leads to more maintainable products. In our case, it was very easy to alter design decisions after we had built RL: we simply redefined type equations (to alter the implementation of its containers) and recompiled; no other source code was changed.

Conclusion

Component-Based Software Development has a lot of benefits over conventional one of developing more reliable and efficient large-scale and complex software systems. Definitely, Component-Based Software Development approach will be the future of software development process to serve the rising demands of software products. In this paper, we have discussed in details about the benefits of Component-Based Software Development over conventional approach and also shown some comparison results using few practical examples. In future one can try to analyze both these approaches more seriously and give some fruitful and systematic directions on these evolutionary fields as well.

References

Fayad, M. E. and Schmidt, D. C., 1997. Object-Oriented Application Frameworks, CACM, 40(10).

Lazar, B., 1998. IBMs, San Francisco Project, Software Development, 6(2): 40-46.

Souza, D. F. D. and Wills, A. C., 1998. Objects, Components and Frameworks with UML: The Catalysis Approach, Addison Wesley, http://www.iconcomp.com.

Ning, J., 1996. A Component-Based Software Development Model, Proc. COMPSAC, pp. 389-394.

Aoyama, M., et al, 1998. An Architecture of Software Commerce Brokerover the Internet, Proc. WWCA (Worldwide Computing and Its Applications) 98, LNCS,1368(1):97-107.

Smith, D.R., 1990. KIDS: A Semiautomatic Program Development System, IEEE Transactions on Software Engineering , pp.1024-1043.

Batory, D., Barnett, J., Garza, J., Smith, K., Tsukuda, K., Twichell, B., and Wise T., 1988. Genesis: An Extensible Database Management System, IEEE Transactions on Software Engineering, pp.1711-1730.