Business-Oriented Component-Based Software Development and Evolution

Stan Jarzabek[1]
Dept. of Information Systems & Computer Science
National University of Singapore
stan@iscs.nus.edu.sg

Martin Hitz
Inst. f. Angew. Informatik & Informationssysteme
University of Vienna
Martin.Hitz@UniVie.ac.at

Abstract

Huge size and high complexity of legacy software are the main sources of today's software evolution problems. While we can ease software evolution with re-engineering tools, in the long term, we should look for a more fundamental and effective solution. Component-based software development (CBSD) technology makes it possible to build software systems as collections of cooperating autonomous application components. This new paradigm has a potential to ease software evolution problems as modification or replacement of components is deemed to be much easier than modification of today's huge monolithic legacy programs. For CBSD to bring promised benefits, we must identify the right components in a given business domain. The claim of this paper is that while CBSD is an important enabling technology, the decomposition of a software system into components must be driven by business considerations. If we let logical models of business processes drive planning and design of software systems, we can avoid creating huge legacy software. Similar approaches may apply to software evolution in other than business domains, too.

1. Introduction

Software evolution takes many different forms ranging from fixing errors to implementing functional enhancements, conversions of programs to other platforms, restructuring for ease of maintenance and re-engineering. Evolution of business environments and technological changes affect software requirements and trigger the need for software evolution. However, the pace of software evolution appears to be much slower than the pace of changes in business and technology. As a result, many of today's programs fall short of user expectations. Increasing costs of adapting software to ever changing business needs and technology (industrial surveys indicate increasing maintenance costs, mounting to 80% of computing budgets [16]), makes companies consider software re-engineering as a cure to maintenance problems. While successful re-engineering projects have been reported [19,20,1], the same sources inform us about the risks and high failure rate of reengineering projects. Reengineering tools, in hands of experienced programmers, can help in on-going maintenance and reengineering, but they do not provide an ultimate solution to software evolution problems. Finally, there is no guarantee that modern CASE tools and visual development environments will help us avoid similar problems in the future, either.

Emergence of component-based software development (CBSD) technology in recent years brings a new hope. Industry interest in component technologies such as ActiveX™, COM™, CORBA™ or JavaBeans™ grows rapidly. It is hoped that with the CBSD technology, we can build software systems out of reusable and autonomous application components with well-defined functionality and interfaces. Software evolution can then become an easier task, as we can evolve software systems by modifying or replacing application components. Modification of components would have only local impact, therefore could be implemented fast. Finally, with CBSD, we could reuse component applications in different software systems.

While software technology for CBSD becomes already available, it is illusory to think that the component technology alone will free us from software evolution problems. In particular, the critical question of "what should constitute an autonomous component in a software system so that software evolution is easier?" cannot be answered by the CBSD technology alone, as the answer to this question depends primarily on business considerations. Software evolves because it is an integral part of a larger, also evolving business system: 50% of maintenance costs are due to program enhancements in response to changes in business requirements [16]. The claim of this paper is that only from the business perspective can we correctly understand, formulate and solve current software evolution problems. The CBSD is an important enabling technology that will help to put business-oriented and component-based solution into practice.

Lack of business-orientation during software planning and development is responsible for problems with today's legacy software systems. Over decades of computerization,

[1] This work was supported by National University of Singapore Research Grant RP3950616.

many companies have been developing software in an ad hoc way rather than according to a business plan. Due to concentration on local needs of company departments and lack of attention to the global business needs, the business value of software often fell short of expectations: many software systems did not address central issues such as using software to automate and smoothen business processes and supporting information needs of the corporate management [7]. In response to those problems, software systems have been heavily maintained, producing huge, degrading in quality, overloaded with redundant functions and difficult to evolve legacy software.

We try to save legacy software with tools that improve code quality through re-engineering or help understand complex programs through reverse engineering and static program analysis. While these tools can help to some extent, in the long term, we should look for more fundamental solutions to software evolution problems. Instead of struggling with evolution of huge and complex software systems, we should ask how we can avoid creating huge systems in the first place. In this paper, we argue that business-oriented criteria for component identification together with domain analysis, generic software architecture design techniques and the CBSD technology can bring substantial improvements to software evolution.

Sections below discuss component technology and software evolution in the context of large companies, such as banks or manufacturing, that develop software to run their own business. At the end of the paper, we argue that discussed concepts apply in other software development situations and in non-business domains, too.

2. Business perspective on software components

In response to software problems, in the 1980s methods for strategic planning emerged, James Martin's Information Engineering [14], IBM's Business System Planning and Zachman's enterprise information architecture [7,21], to name a few of them. The objective of strategic planning methods is to provide an element of coordination and business-orientation to software development. Among these methods, Zachman's approach has some unique features that make it attractive in the context of the CBSD technology and software evolution.

The aim of the enterprise information architecture is to provide a stable, business-oriented framework for software development in a company. An enterprise information architecture has a hierarchical structure, with different levels reflecting views of different stakeholders. It consists of three vertical columns (data, process and technology) and six horizontal levels. The upper two levels (ballpark and owner's views) address business concerns. The lower levels define information systems a company needs to develop, how they should be fragmented into smaller applications to avoid redundancy, what applications should do to add value to the whole company in terms of support for both cross company business processes and information needs of decision makers, which applications should work together to support higher level functions and, finally, how applications should be built and installed. We refer the reader to [7,21] for more details of the enterprise information architecture.

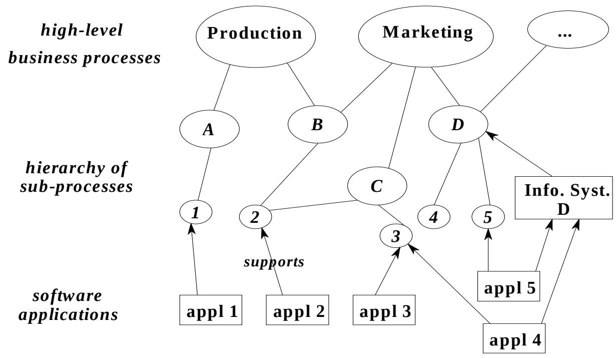

Figure 1. An enterprise information architecture

Fig. 1 illustrates a sample fragment of an enterprise information architecture. Production and Marketing business processes have been decomposed into lower level processes (A, B, C, and D in circles) which in turn have been decomposed into elementary processes (smaller circles labeled with digits) roughly corresponding to *use cases* as described in [9]. High level processes in Fig. 1 correspond to end-to-end cross-departmental business processes. They are modeled as chains of elementary processes. We call processes in the enterprise information architecture *logical processes*, as we derive them from models of what a company does rather than from how a company does it in terms of company organizational charts.

The above information architecture is a prelude to building evolving business software. We start by developing small-size software applications to support elementary processes (circles with digits). Then, we compose larger information systems, supporting higher-level processes, out of small applications. In that way, software structure is closely integrated with the structure of business processes. The impact of changes in a business environment can more easily be traced to the level of software. As business evolves, software can evolve accordingly. Re-engineering of a business processes will usually affect some of the related applications but will not require massive changes (or scrapping) of the existing information systems. As logical business processes are independent of the company structure, we can also change the company structure and evolve information systems by re-configuring component applications to fit into the new structure.

Although the concept of an enterprise information architecture has been known for some time, it has not had much impact on the structure of business software and did not save us from problems of maintaining today's legacy software. We could blame lack of education, mismatch between the architectural concepts and organizational structure of companies and many other non-technical issues. However, it also true that only with the recent introduction and acceptance of the CBSD technology standards, software component solutions suggested by the enterprise information architecture can be directly and easily implemented. Properly identified functionalities at the business level can be implemented as stand-alone executable applications, encapsulated as COM objects, with an interface providing access to application functions. Keeping applications small and the ability to use them in different configurations is now possible and simple.

3. Software decomposition for ease of evolution

Long ago, Parnas noticed that, for any given problem, there are many possible ways to design a program solution [18]. A required program structure depends on the objectives that we want to achieve such as simplicity of the design, run-time efficiency, ability to change design decisions, portability across platforms or ease of enhancing program functions. Here, we are interested in how we should decompose software systems into a set of executable application components for ease of evolution. Obviously, there are many ways we can do this. The lesson from the discussion in the previous section is that to ease software evolution we should structure business software into components according to business criteria. Strictly speaking, *software components should correspond to business processes at different levels of an enterprise information architecture*. The interactions between business processes should determine interfaces between software components. Functionally coupled components should be packaged into one application and functionally coupled applications into an information system. Components that share much data should be built around common databases, ideally conforming to a business wide conceptual data model.

4. A model for component-based software evolution

Consider the following situation. Suppose we identified a business process P that consists of a chain of sub-processes $p_1,\ldots,p_n$. Software applications App(p_i) support sub-processes p_i and an information system Is(P) supports the whole process P. Evolution of applications App(p_i) may include any of the following activities:

P1. modifying the functionality of an application App(p_i) through:
   a. customization mechanism provided by the CBSD framework such as COM, ActiveX, etc.,
   b. modifying code of the App(p_i),

P2. modifying interfaces between App(p_i) and other applications using the CBSD framework,

P3. modifying interfaces between App(p_i) and databases.

Evolution of Is(P) may include any of the following activities:

S1. evolving one or more component applications App(p_i), as described above,

S2. adding, deleting, replacing component applications,

S3. modifying the part of Is(P) that does not belong to any of its component applications. For example, the overall control of Is(P) is likely to be implemented within Is(P), outside the component applications.

In the above evolution model, we treat applications as either black-box components equipped with vendor-provided customization mechanism (options P1a, P2 and P3) or as open components that can be customized by code modifications (option P1b). Such a view is simplified and may not allow us to exploit all potentials of the CBSD paradigm. Now let us examine how program construction time reuse facilitated by domain analysis and generic software architectures can enhance the above model and better support software evolution.

5. Generic software architectures, CBSD and software evolution

In Fig. 1 we see that some of the lower level processes recur in more than one higher level business processes. For example, process 2 occurs in processes B and C, while process B occurs in Production and Marketing. Furthermore, when deciding which departments will be performing which processes, we can find out that many departments may deal with different instances of the same process. For example, process ‘procurement’ is likely to appear in many high-level business processes and in many departments, even if only a subset of functions are performed there.

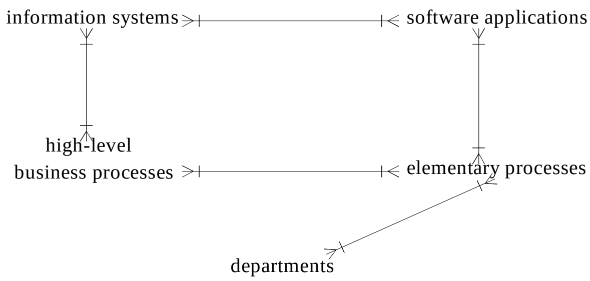

Figure 2. Many-to-many mappings between entities in the enterprise information architecture

Fig. 2 depicts these many-to-many relationships among business and software entities in Martin/Odell notation [15]. A cost-effective strategy for software development and evolution should not ignore an opportunity to reuse software applications that recur in more than one context.

An enterprise information architecture helps us identify reuse opportunities. However, not always is it easy to tap reuse opportunities, even though we know they exist. Apart from similarities, there will be differences among processes and applications that recur in different contexts. Consequently, there will be differences across software applications that support those processes. Applications may vary in user requirements or attributes of data. Different applications may run on different platforms, use different DBMSes, etc. To reuse, we need implement generic applications that can be customized to various contexts in which applications are to function.

The CBSD paradigm can be practical only if software components are highly customizable - does the CBSD technology provide a sufficiently strong mechanism to create such applications? Technologies such as COM and ActiveX partially address this issue by providing a mechanism for run-time customization of application components. We can view applications as objects with rich interfaces providing access to different functions of an application. User-defined active control objects provide selective access to application functions and allow us to further customize the way applications interact one with another. However, there is a limit to such black-box customization and there will be cases when the CBSD technology will fail to provide a sufficient customization mechanism. In the evolution model described in section 4, we must customize an application by modifying its code (option P1b) for such cases. Code modifications are difficult, time consuming and over time degrade the quality of an application. To better address the issue of application component customization, we should view executable applications as structured objects, built themselves out of generic components customizable at program construction time.

Based on the above observations, we distinguish the following two types of software components:

1. executable components such as applications or software systems (sets of co-operating applications), and
2. generic software components that are used to build applications at the program construction time.

The construction time components should provide a mechanism for customizing executables to a wide range of variant requirements, in particular to those variants that the CBSD technology cannot address. These construction time generic components will allow us to limit the need for modifying components at the code level, easing software evolution. Finally, construction time components should be reusable across a family of similar applications.

Run-time and construction time components and customization mechanisms complement each other. Understanding how they can work together is essential to deliver easy to use, flexible and evolving component-based solutions.

The reuse community coined the term *Domain-Specific Software Architecture* (DSSA) for techniques dealing with designing generic architectures in specific domains such as procurement [6]. Processes in the enterprise information architecture, especially ones that appear at lower levels in Fig. 1, are similar to *domains* in the DSSA approach. Instead of building a new software application for each instance of a generic process, we can build a common, generic architecture for a family of applications first, and then develop specific applications by customizing a generic architecture.

From a business perspective, software reuse is closely related to software evolution. At the technical level, reuse even further reinforces software evolution by limiting redundant code. Generic software functions are implemented within generic software architectures. During software evolution, we modify a generic architecture and changes are automatically propagated to all software applications that are based on that architecture.

6. A refined model for component-based software evolution

A model sketched in this section shows how the CBSD and DSSA technologies can work together to foster software reuse and ease evolution. Consider the same situation as in our initial software evolution model (section 4). But this time, we assume that (some of the) software applications are built based on generic software architectures. Suppose an application App(p_i) is an instance of a generic architecture Gen(p_i). Evolution of application App(p_i) may include any of the following activities:

G1. modifying the functionality of an application App(p_i) through:
  a. a customization mechanism provided by the CBSD framework such as COM or ActiveX,
  b. customizing Gen(p_i),
  c. evolving Gen(p_i),
  d. modifying App(p_i) code in ad hoc way,

G2. modifying interfaces between App(p_i) and other applications,

G3. modifying interfaces between App(p_i) and databases.

The evolution model for information systems is the same as in section 4. Options G1a-c are more systematic and cost-effective ways of evolving software than option G1d. Option G1a refers to the run-time customization of application components. Option G1b reflects small evolution steps that have been envisioned during the design of a generic architecture. Occasionally, we shall inevitably face more substantial, new software requirements that cannot be addressed by means of customizing a generic architecture. Option G1c reflects the situation when we decide to evolve the architecture to accommodate these new requirements as an integral part of the architecture.

7. Software evolution in non-business domains

At first look, it may seem that the above software evolution concepts are only relevant to large, multi-department companies that develop software for in-house use. we would like to argue that similar concepts also apply to software evolution in non-business domains.

Domain analysis facilitates development of product families in both business and non-business domains [14,18]. Many software houses develop programs that differ in user requirements or run on different platforms but otherwise display many similarities. A program family is based on a generic architecture that implements features common to all family members. This is in contrast to the situation whereby we treat each program as a separate, unique entity that is developed in isolation from other programs. Concepts of program families are relevant to most application domains.

Domain analysis [2,3,4,5,19], an activity that leads to identifying commonalties and differences across program family members, bears many similarities to building an enterprise information architecture. While during building an enterprise information architecture we identify processes that can be supported by small applications, during domain analysis we identify software functions, design patterns and code that can be supported by self-contained software components. In the same way as an enterprise information architecture facilitates evolution of business software, domain analysis can facilitate software evolution in other domains.

The emerging CBSD technologies facilitate building software systems out of relatively small, problem-oriented application components. CBSD offers mechanisms for run-time customization and integration of such components. To identify and, eventually, build such components, we need a proper decomposition technique for domain analysis [11,12,13]. This technique should facilitate creation of partial problem-oriented domain models and generic architectures for problem-oriented software components. In such a scenario, software applications would be created by customizing and combining generic architectures. Software evolution would be done by run-time customizations of application components or program construction time customization of generic architectures rather that by patching code in an ad hoc way. Combining the CBSD and generic software architecture concepts seems to be a natural way to advance software evolution, reuse and software engineering practice in general.

What are the criteria for decomposing a domain so that we observe the above benefits? We think decomposition should identify problems that can be considered instances of the same generic problem [8]. Examples of such problems are process monitoring, failure tracking, metric collection, etc. Problems that occur in many software applications, possibly in different application domains, are most important. Problem-oriented domain components may be identified at user requirements, design and implementation levels.

8. Conclusions

We presented a view that software decomposition based on business criteria leads to software that easily evolves. The emerging CBSD technologies and generic software architectures complement each other by providing run-time and construction time component customization mechanisms, respectively. They both contribute to component-based software systems that can evolve easily and whose quality will not degrade over time. we argued that generic software architectures reinforce software evolution, as some of the software evolution activities can be done at the software architecture level rather than by patching code in an ad hoc way. Finally, we suggested that, in non-business domains, problem-oriented domain analysis similar to enterprise information architecture modeling, leads to identifying components of a software architecture that can easily evolve.

References

[1] Adolph, W.S. "Cash Cow in the Tar Pit: Reengineering a Legacy System," *IEEE Software*, May 1996, pp. 41-47

[2] Arango, G. “Domain Analysis - From Art Form to Engineering Discipline,” *Proc. Fifth Int. Workshop on Software Spec. & Design*, May 1989, Pittsburgh, pp. 152-159

[3] Bassett, P. *Framing Software Reuse - Lessons from Real World*, Yourdon Press, Prentice Hall, 1997

[4] Batory, D et al. “The GenVoca Model of Software-System Generators,” *IEEE Software*, September 1994, pp. 89-94

[5] Biggerstaff, T. and Perlis, A. (Editors) *Software Reusability*, vol. I and II, ACM Press, 1989

[6] Coglianese, L, Tracz, W. et al. Collected Papers of the Domain-Specific Software Architectures (DSSA) Avionics Domain Application Generation Environment (ADAGE) Project, ADAGE-IBM-93-09A, July 1994

[7] Cook, M.A. Building Enterprise Information Architectures - Reengineering Information Systems, Prentice Hall, 1996

[8] Fowler, M. Analysis Patterns: Reusable Object Models, Addison-Wesley, 1997

[9] Jacobson, I. Object-Oriented Software Engineering, Addison-Wesley, 1992

[10] Jarzabek, S. “Modeling Multiple Domains for Software Reuse,” *Symposium on Software Reusability, SSR’97*, Boston, May 1997, ACM Press, pp. 65-79

[11] Kang, K.S. Cohen, et al. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*, Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, November 1990

[12] Lung, C. and Urban, J. “An Approach to the Classification of Domain Models in Support of Analogical Reuse,” *ACM SIGSOFT Symposium on Software Reusability*, *SSR'95*, Seattle, 1995, pp. 169-178

[13] Macala R. et al. “Managing Domain-Specific, Product-Line Development,” *IEEE Software*, May 1996, pp. 57-67

[14] Martin, J. *Information Engineering*, Prentice-Hall, 1986

[15] Martin, J. and Odell, J. Object-oriented analysis and design, Prentice-Hall, 1992

[16] McClure, C. The Three Rs of Software Automation: Re-engineering, Repository and Reusability, Prentice Hall, Englewood Cliffs, NJ, 1992

[17] Parnas, D. “On the Design and Development of Program Families,” *IEEE TSE*, March 1976, pp.1-9

[18] Prieto-Diaz, R. “Domain Analysis for Reusability,” *Proc. COMPSAC’87*, 1987, pp. 23-29

[19] Sneed, H. “Planning the Reengineering of Legacy Systems,” *IEEE Software*, January 1995, pp. 24-34

[20] Ulrich, W. "Re-development Engineering: Formulating an Information Blueprint for the 1990's," *CASE Outlook*, No. 2, 1990, pp. 15-23

[21] Zachman, J. “A Framework for Information Systems Architecture,” *IBM Systems Journal* 26, no. 3, 1987