Component Criteria for Information System Families

Stan Jarzabek

Department of Computer Science, School of Computing
National University of Singapore
Lower Kent Ridge Road
Singapore 119260
stan@comp.nus.edu.sg

Abstract. In this paper, we discuss component technologies in the context of information system (IS) families. An IS family is characterized by common requirements, shared by all the family members, and variant requirements that may differ across family members. Many variant requirements are non-local, i.e., they cannot be confined to a single system component, on contrary, they affect many components in complex ways. An effective generic architecture for an IS family should provide means to handle anticipated and unexpected variant requirements and support evolution of the family over years. In the paper, we illustrate problems that arise in supporting IS families and describe a generic architecture that includes global, cross-component structures to deal with changes during customization and evolution of an IS family.

1 Introduction

An information system (IS) family comprises similar information systems. A family may address a coherent business area (such as payroll or customer order processing) or a certain problem area (such as task management or failure detection). Members of an IS family share common characteristics, but also differ in certain variant requirements. A systematic way to support an IS family is to implement common and variant functions within a generic architecture and then develop individual information systems by customizing and extending the architecture. Many variant requirements for IS families are non-local, i.e., they affect many system components. An effective generic IS architecture should help developers in handling non-local variant requirements. Furthermore, to support evolving business, a generic IS architecture itself must evolve by accommodating new requirements. For the long-term success of an architecture, methods and tools should be provided to keep the complexity of a growing architecture under control. In practice, customizations and evolution of IS family architectures appear to be difficult tasks.

In recent years, Component-based Software Engineering (CBSE) and Distributed Component Platforms (DCP) [6] have received much attention. Component-based systems are built out of autonomous, independently developed runtime components. By conforming to the DCP's interoperability standards, functionality of a component-based system can be extended either by adding new components to the system or by replacing a certain component with another one, providing richer functionality. Binary components can be customized using the introspection facility [6]. Component technologies promise to better facilitate reuse and improve software development and maintenance productivity.

In this paper, we shall discuss component technologies in the context of IS families. In our experience, generic IS architectures based on runtime components may not provide sufficient support for customization and evolution. We shall illustrate the problem with an example from our project and outline a possible remedy to the problem.

2 Related Work

The concept of program families was first discussed by Parnas [7] who proposed information hiding as a technique for handling program families. Since then, a range of approaches have been proposed to handle different types of variations (for example, variant user requirements or platform dependencies) in different application domains. Pre-processing, PCL [8], application generators [3], Object-Oriented frameworks [5], Domain-Specific Software Architectures [9], frame technology [2] and, most recently, distributed component platforms – they all offer mechanisms for handling variations that can be useful in supporting IS families.

A software architecture is described by a set of components, externally visible properties of components and component relationships [1]. The rationale for most of the architectures we build is to facilitate reuse – with possible modifications, we wish to reuse an architecture in more than one project. Therefore, most of the architectures really underlie system families rather than a single system. A number of authors advocate clear separation of a construction-time software architecture from it's runtime architecture [1,2,4]. The major concern of construction-time architectures is flexibility, i.e., the ability to customize components to meet variant requirements of products and the ability to evolve the architecture over time to meet changing needs of the business environment. Issues of interest in runtime architectures include allocation of functions to components, deciding which logical components should be packaged into a single executable, parallel execution of components, data communication, invocation of services between components, overall control and synchronization. A generic IS architecture is a software construction-time architecture for supporting an IS family. In a generic IS architecture, some of architecture components may be optional, incomplete or missing. During customization, developers select components and customize them to accommodate specific variant requirements to be implemented in the target information system.

In frame technology [2], a software construction architecture consists of a hierarchy of generic components called frames. A frame is a text (written in any language, such as, for example, IDL™ or Java™) with breakpoints. Frames can be adapted to meet variant requirements of a specific system by modifying frame's code at breakpoints. Frames are organized into frame hierarchies and all the modifications needed to satisfy given variant requirements can be traced from the topmost frame in the hierarchy, called a specification frame. A frame processor is a tool that customizes a frame hierarchy according to directives written in the specification frame and assembles the customized system.

Boca [5] provides a meta-language to define business semantics as a central part of the construction-time architecture. Business components such as customers, orders, employees, hiring and invoicing are specified in the meta-language, separately from the runtime program characteristics. A meta-language provides means for maintaining integrity of requirements for a system family during customization and evolution. Boca supports synthesis of component-based runtime systems from business and implementation-specific component layers. A construction architecture makes it possible to separate business concerns from platform concerns.

3 A Generic IS Architecture Based on Runtime Components

Consider a family of Facility Reservation Systems (FRS). Members of the FRS family include facility reservation systems for offices, universities, hotels, recreational and medical institutions. There are many variant requirements in the FRS domain: In some cases, an FRS should allow one to define facility types (such as Meeting Room and PC) and only an authorized person should be allowed to add new facility types and delete an existing facility type; some FRSes may allow one to view existing reservations by facility ID and/or by reservation date.

During runtime, we want FRSes to consist of three tiers, namely user interface, business logic and a database. Each of these tiers is a component that consists of smaller components, implemented according to standards of the underlying platform (for example, EJB™ or ActiveX™). Suppose we decide to manage an FRS family in terms of runtime components, using one of the available visual environments. A developer should be able to selectively include variant requirements into a custom FRS. To achieve this, all the anticipated variant requirements should be implemented within the relevant components. Most often components affected by variant requirements span all three tiers. A developer might use an introspection facility [6] to customize components. He or she would set component property values to indicate which reservation viewing methods are needed in a target FRS. After customization, components would reveal only the required functions. In the situation of an IS family, this scenario may not work well. Each family member we build will have to include implementation of all the variants, even though some of these variants will never be used. As more and more variants are implemented, components will grow in size. Also, keeping track of how variants affect components must be taken care by developers. Adding new unexpected requirements to the FRS family is not easy, either. If the source code for components affected by new requirements is not available, there is no simple way to implement a new requirement. Developers may need to re-implement affected components, as components cannot be extended in arbitrary ways without the source code. If the source code for relevant components is available, developers could use either inheritance or a suitable design pattern to create new components that would address the new requirement. Visual environments built on top of DCPs support the former solution and some support the latter one (see, for example, the San Francisco framework http://www.ibm.com/Java/SanFrancisco, chapter 8 "Application Development methodology"). While this method of addressing new requirements is sufficient in the rapid application development situation, it presents certain dangers in the context of generic architectures for IS families. Over years of evolution, an architecture may be affected by many new requirements. Implementation of new requirements will add new components (or new versions of old components) to the architecture. As certain requirements may appear in different combinations – we may end up with even more components. As a result, our architecture may become overly complex and difficult to use. These components, growing in size and complexity, have to be included into any information system built based on the architecture, independently of whether the options are need or not. In long-term, accumulative result of this practice is likely to be prohibitive.

4 A Construction-Time Extension of a Runtime Architecture

An architecture based on runtime components does not allow us to exploit software flexibility to its fullest potential [2]. We can alleviate the problems discussed in the last section by designing a generic IS architecture based on construction units that facilitate change better than runtime components. A generic architecture should make it clear how to handle variant requirements and provide a systematic way of extending IS family with new unexpected requirements. While runtime components must be complete executable units, construction units of a generic architecture may be parametarized by variant requirements and may require pre-processing before they can be compiled. Furthermore, construction units may be incomplete in the sense that we may encapsulate different aspects of a system such as business logic, platform-dependencies (e.g., event handling code), etc., in separate units. A proper tool will automatically apply a composition operation to combine required construction units into components of a custom system. In the above scenario for supporting an IS family, the emphasis is shifted from ready to use components to the process that produces components from construction units on demand. We think this is essential to keep the complexity of the architecture under control. By studying the customization process for anticipated variant requirements, developers can also better understand how to deal with unexpected requirements that arise during system evolution.

Frame technology [2] directly supports most of the above concepts. It forms a construction environment for managing system families, in particular, it can be applied to component-based systems. In our domain engineering project, we designed a generic FRS architecture as a hierarchy of frames, where frames correspond to components of the FRS runtime architecture. We extended a frame hierarchy with an explicit model of commonalties and variations in the FRS domain, and with a Customization Decision Tree (CDT). A CDT helps understand customizations that lead to satisfying variant requirements. Nodes in the CDT correspond to variant requirements. A script attached to a CDT node specifies customizations of a generic architecture for a given variant requirement. Frame processing is based on a composition operation that can be applied at different levels of abstraction. Before applying a frame processor, we assemble customization scripts from the CDT into a specification frame. The frame processor interprets the specification frame to produce a custom software system, according to a blueprint of its component-based runtime architecture.

5 Conclusions

In the paper, we discussed problems that arise when we base a generic architecture for an IS family on runtime components. As customizations and evolution of an IS family most often affect many runtime components, it is better to base a generic IS architecture on construction units that are designed for the purpose of dealing with changes. Construction units should be parameterized by variant requirements and should partition the implementation space into cohesive, manageable parts that can be combined into custom components using a composition operation and a suitable tool.

Acknowledgments

Thanks are due to Cheong Yu Chye who implemented a generic architecture for the FRS family. I am indebted to Paul Bassett for letting us use the Netron's product Fusion for our projects.

References

1. Bass, L., Clements, P. and Kazman, R. *Software Architecture in Practice*, Addison-Wesley, 1998
2. Bassett, P. *Framing Software Reuse - Lessons from Real World*, Yourdon Press, Prentice Hall, 1997
3. Batory, D et al. "The GenVoca Model of Software-System Generators," *IEEE Software*, September 1994, pp. 89-94
4. Digre, T. "Business Component Architecture," *IEEE Software*, September/October 1998, pp. 60-69
5. Johnson, R. and Foote, B. "Designing Reusable Classes," *Journal of Component-Oriented Programming*, June 1988, Vol.1, No.2, pp. 22-35.
6. Krieger, D. and Adler, R. "The Emergence of Distributed Component Platforms," *IEEE Computer*, March 1998, pp. 43-53
7. Parnas, D. "On the Design and Development of Program Families," *IEEE Trans. on Software Eng.*, March 1976, p. 1-9
8. Sommerville, I. and Dean, G. "PCL: a language for modelling evolving system architectures," *Software Engineering Journal*, March 1996, pp.111-121
9. Tracz, W. Collected overview reports from the DSSA project, Technical Report, Loral Federal Systems – Owego. (1994).