Reuse Enabling Technology
On Constructing Systems From Large-grained Components

Wojtek Kozaczynski

Andersen Consulting

Center for Strategic Technology Research

100 S. Wacker Dr., Chicago IL 60606

Tel: 312-507-6682

Email: wojtek@andersen.com

Fax: 312-507-3526

Abstract

This position paper addresses the technological requirements for domain-specific reuse of large-grained software objects. Large-grained software objects are on the order of subsystems or self-sustained modules and we assume that they encapsulate well defined functionality and have formally specified interfaces. Our main concern is what technology must be in place in order to support a development process where complex, distributed systems are composed/assembled from such large-grained components. We briefly describe what we call the Module Development, Coordination, and Interconnection Technology. We argue that this technology should support a system development process that naturally promotes and enables reuse.

Keywords: reuse process, technology, specifications, problem domain

Workshop Goals: present an approach to enable reuse, obtain feedback, learn, networking

Working Groups: process models, tool and environments, domain analysis, management and economics

1 Background

Andersen Consulting is one of the largest consulting companies in the world. A very large part of its revenue comes from building computer systems that support clients' mission-critical business operations. Andersen Consulting perceives great potential value in implementing a large-scale reuse program. A software reuse program can be looked at from three major perspectives: (i) the management infrastructure (investment, dedicated resources/organizations, enacting the process, incentive structure, legal issues, ...) (ii) the professional skills and culture (polarization of skills, new skills development, attitudes, education and training, ...), and (iii) the technical infrastructure.

This paper addresses the third perspective of reuse; the enabling technology. We argue that in order to achieve significant gains from reuse, the process of building systems must be changed. Specifically, the systems should be assembled from large, ready to use components/modules encapsulating well-defined, domain-specific functionality. There is nothing new in this idea except that we assume that:

- it should be possible to develop components independently from each other, using languages that best suit their functionality, and let them execute in environments that best suit their non-functional requirements (including cost)

- it should be possible and easy to assemble components in a number of different configurations without need to change their internals, and

- it should be possible to interconnect the components into a running system despite their heterogeneity (different languages and execution environments).

To be able to do the above a new technology must be produced, integrated, and instrumented with tools. We refer to it as Module Development, Coordination, and Interconnection Technology. This technology, addressed in more detail in the following section, should enable a domain architecture specific, component-based process of building large, distributed systems. Before we proceed, we should make two points explicit.

Point one is that the reuse we are interested in can be characterized as reuse of large-granularity, or large-object reuse. Small-granularity reuse would be the reuse of generic components independent of the problem domain. Examples of small-grained components would be file/DB access functions, data structure manipulation functions, individual object classes, I/O objects and functions, etc. The parts we are interested in are on the order of complete functional modules or subsystems. Examples of such modules would be an order processing subsystem, a customer account maintenance module, an entire UI client, or a production simulation module.

The above point leads to an observation on the reusability of large-grained components. These components encapsulate very domain-specific knowledge and behavior. Due to their size and potential internal complexity, their development may be costly. This cost can be amortized only if the components are used repetitively with no (or only minimal) changes. Fortunately, Andersen Consulting typically constructs many instances of a generic system type (eg. inventory management) in the same domain for different clients. Moreover, the company is organized along specific business or industry domains (banking, insurance, utilities, ...). This simplifies the facilitation and monitoring of a domain-specific reuse process.

2 Position

The Software Engineering Lab of the Center for Strategic Technology Research (CSTaR) is evaluating the feasibility of producing the Module Development, Coordination, and Interconnection Technology. The technology should support a system building process that naturally promotes reuse by implementing the principles of separation of concerns, abstraction, and functional decomposition. The process, illustrated Figure 1, can be described as follows:

Figure 1. Domain-specific, component-based system development process abstractions and products.

- The interfaces of a reusable module are formally specified in an interface specification language. Such a specification describes the services provided and required by the module and the conditions under which they can be rendered. The specification becomes a contract between module developers and module users and separates their concerns. Modules are developed with minimal assumptions of how they will interconnect with the other modules they will request services from or provide services to. They can be written in a number of languages for which the bindings (with the interface spec language) are defined and can run on a set of predefined platforms (computing environments).

- Module interface specifications (not the modules implementations) are used to develop a specification of module coordination and cooperation, and assign modules to computing resources. The designer (a person who composes the system from reusable modules) works with a set of abstractions of communication services and computing resources rather than with particular services or resources of an execution environment. These abstractions are provided by the Connection Infrastructure Formalism. They are a boundary between his concerns and the concerns of the software engineer who will implement the execution environment(s) for the system. Examples of these abstraction are: 1-to-1 synchronous communication, 1-to-n asynchronous communication with the "all must receive" requirement, process, etc.

- The abstractions of the connection infrastructure are mapped into a number of different implementations of execution environments. For example, the underlying execution environment can be Unix-based and use only RPCs to support communication between modules. On the other hand, it may be a proprietary software bus like Andersen's FCP (FOUNDATION for Cooperative Processing) that runs on a network of different computers and workstations.

- The final run-time version of a system is assembled/made from the functional modules (the reusable, domain-specific modules), modules developed to help coordinate the work on the functional modules (the control modules), automatically generated module adapters, and the underlying execution environment services. The only part that we may not be able to generate from the specifications (as shown in the Figure) are the control modules. It would be naive to assume that a complex system can be assembled entirely from reusable modules:

    - If functions are missing, new modules (hopefully reusable) must be developed or existing ones must be modified effectively giving rise to new modules or versions.

    - In order to support complex, multi-module interactions, control modules may have to be written. These modules should have no domain-specific function but coordinate the work of the functional modules.

We strongly believe that the technology to enable the above process can be developed and packaged into tools. The process itself has a number of obvious advantages:

- The development of a few systems in the same (sub)domain should result in a library of reusable modules and system designs (module coordination and cooperation and resource assignment designs). These modules and designs are a very tangible form of a Domain-specific Software Architecture.

- The module development, testing and certification has been separated from system assembly and testing. These activities can be delegated to different groups of specialists and performed at different locations. For example, Andersen Consulting is forming a number of geographically distributed Solution Engineering Centers. Most of the module development and testing will be done at these centers. However, systems assembly and testing will be done by engagement teams at the client locations. Also, even if new modules must be made for a system, their development can overlap in time with system testing. This is simply done by using the module interface specification as a contract and as a base for developing or generating a module stub at the same time.

- System quality and process productivity should naturally increase with every subsequent iteration of a system development. Similarly, the quality of the reusable components and the system designs should increase.

- The decomposition of a systems into relatively independent modules with the interconnection, coordination, and communication logic removed from them should allow for relatively easy construction of large, distributed, and heterogeneous systems.

- The same decomposition of systems into modules with well specified external behavior will enable flexible system adaptation and tuning. For example, a new, better performing version of a module can be developed and put in place of the old version with only minimal, automatically applied changes to the system.

- The approach provides a consistent way of treating and using legacy systems. After a wrapper and an interface specification is developed for a legacy system (or its part), it can be treated as a module.

3 Comparison

CSTaR's Software Engineering Lab is currently conducting three parallel projects that collectively address the technology described in the previous section:

- a project on module interface specifications
- a project on system distributed architecture design (related to the coordination, cooperation and resource allocation design), and
- a project on software buses (related to the interconnection infrastructures).

These projects borrow ideas from a number of other industrial and academic research projects. Similar ideas of module interface specification and flexible module interconnection can be found in the work of Jim Purtilo [6,7] and Dewayne Perry [3,4]. Perry introduces an idea of module service pre- and post-conditions that improve the semantic richness of the module interface specifications. Purtilo introduces the notion of a software bus (an interconnection infrastructure), automatically generated adapters, and application geometry design which is similar to the module cooperation and coordination design. However, both authors are looking at small-grained module reuse rather than large-grained module reuse. Perry also assumes that modules are assembled into a single run-time unit and therefore share address space. We assume explicitly that modules are distributed and communicate only via messages.

The idea of a unifying interconnection infrastructure and an interface specification language is also central to OMG CORBA [2]. CORBA is a standard for the developers of OO software buses. From our perspective a CORBA-compliant bus is one of many possible implementations of an interconnection infrastructure. Some of the abstract services in our connection infrastructure formalism have no equivalent counterparts in the CORBA specification and have to be constructed from its lower-level services. Also, the issues of distributed systems design are conceptually higher than the issues addressed by CORBA.

The ideas of domain-specific software architectures and their role in reuse are obviously not new [5]. However, we have defined domain-specific software architectures very pragmatically as collections of reusable modules and systems designs. On the other hand, it seems intuitive that our reusable component libraries should be organized along domain-specific taxonomies described in a formal way (similar to RLF [1], for example) and should contain not only modules but also other design artifacts.

References

[1] *Technical Concept Document : CARDS,* STARS-C-04107A/001/01, PARAMEX, 1993

[2] *The Common Object Request Broker: Architecture and Specification ,* OMG Document Number 91.12.1

[3] Dewayne E. Perry, "The Inscape Program Construction and Evaluation Environment", Tech. Report, Computer Technology Research Lab., AT&T Bell Laboratories, 1986

[4] Dewayne E. Perry, "Software Interconnection Modules", *Proceedings of the International Conference on Software Engineering*, Monterey, CA, May-April 1987

[5] Ruben Prieto-Diaz and Guillermo Arango, "Domain Analysis and Software System Modeling", *IEEE Computer Society Tutorial,* Los Alamitos, CA 1991

[6] James M. Purtilo, "The POLYLITH Software Bus", University of Maryland CSD Technical Report 2469, 1990

[7] James M. Purtilo, Richard T. Snodgrass and Alexander L. Wolf, "Software Bus Organization: Reference Model and Comparison of Two Existing Systems", DARPA Module Interconnection Formalism Working Group, Technical Note No. 8, November 1991

4 Bibliography

Dr. Wojtek (Voytek) Kozaczynski is the director of the CSTaR's Software Engineering Laboratory. Before assuming this position in 1992 he had been the principal investigator on the software analysis and re-engineering project. The project resulted in development of two experimental workbenches supporting the activities of understanding and design recovery of legacy systems as well as recovery of reusable components from these systems.

Dr. Kozaczynski's research interests include: software development environments, software reuse, software renovation, program analysis and understanding and automatic program transformation. He also has an extensive database background that includes the development of a commercial DBMS and work on adaptive database decomposition in distributed databases. Prior to joining CSTaR in 1988 he was working as an Assistant Professor at the Department of Information and Decision Sciences, University of Illinois at Chicago, where he taught and researched the application of AI techniques to information systems design and development and database design. Dr. Kozaczynski came to the U.S. in 1982 after receiving his graduate degree from the Technical University of Worclaw, Poland.