The Road to Reuse: Design Patterns
Doug Schmidt,
Washington University, St. Louis, MO.
schmidt@cs.wustl.edu

Object-oriented (OO) techniques are a promising means to achieve widespread reuse of software designs and components. Like so many other promising techniques in the history of computing, OO has yet to deliver widespread improvements in software reuse. There have certainly been successes — sophisticated frameworks of reusable components are now available in many OO languages running on many OS platforms. In general, however, these frameworks have been limited to certain well-traveled domains, like graphical user-interfaces. Moreover, component reuse is often limited to 3rd party libraries and tools, rather than being integrated into organizations' internal software development processes.

A number of factors are responsible for the lack of widespread software reuse. Commonly cited impediments to success include:

Organizational factors — such as the effort required to catalog, archive, and retrieve reusable components

Economic factors — such as the lack of adequate chargeback schemes and monetary incentives in many development organizations

Political factors — such as not wanting to share components with rival groups

Psychological factors — such as perceived threat to job security and the ubiquitous "not invented here" syndrome.

These factors are often tied to deeply rooted problems in corporate culture, which is hard to change. In a very real sense, these problems transcend the scope of OO and any other methodology short of mandatory 12 step programs...

Fortunately, there are ways that OO design and programming techniques can, and have, had a powerful impact on improving software reuse. In general, OO techniques facilitate reuse by enabling larger grained abstractions and encapsulation mechanisms than earlier paradigms (such as procedural design). OO practitioners have traditionally focused on the design and implementation of reusable components in terms of artifacts such as classes, class categories, objects, modules, and subsystems.

Unfortunately, many OO reuse efforts have focused on language features (such as inheritance, polymorphism, generics, and exception handling) or design methods and notations (such as Booch vs. OMT vs. Shlaer/Mellor). I believe the quest for solutions in languages and methods is alluring because it appears to absolve us from having to address the tough issues. Chief among these is capturing and articulating the abstractions and components required to build widely reusable software in complex domains (such as avionics, business data processing, telecommunications, online transaction processing, and distributed communication systems).

A vexing challenge for domain analysts is translating their domain expertise into reusable software components. A number of approaches (such as transformational systems, expert systems, and domain-specific software architectures) have been advocated over the past decade in both in academia and industry. Few, if any, of these approaches have achieved widespread success. A major impediment is that many of these approaches don't integrate well into the development processes (or lack thereof) found in contemporary software organizations. Most organizations are staffed by developers who are not fluent in formal methods and who are often deeply suspicious of automated techniques that challenge their artisan view of the software craft.

In addition, traditional approaches to domain analysis have often ignored the fundamental challenges in large-scale software system development. These challenges include communication of architectural knowledge among developers; accommodating new design paradigms or architectural styles; resolving non-functional forces such as reusability, portability, and extensibility; and avoiding development traps and pitfalls that are usually learned only by experience.

Design patterns are a promising technique for achieving widespread reuse of software architectures by directly addressing fundamental software development challenges outlined above. Design patterns capture and articulate the static and dynamic structures and collaborations of components in successful solutions to problems that arise when building both general-purpose and domain-specific software. Patterns aid the development of reusable frameworks by expressing the structure and collaboration of components in a software architecture at a level higher than (1) source code or (2) object-oriented design models that focus on individual objects and classes. Thus, patterns facilitate reuse of software architecture, even when other forms of reuse are not feasible (e.g., due to fundamental differences in operating system mechanisms or programming language features).

Patterns help to alleviate software complexity at several phases in the software lifecycle. Although patterns are not a software development method or process, they complement existing methods and processes. For instance, patterns help...
Organization Domain Modeling: A Tailorable, Extensible Framework for Domain Engineering

Mark A. Simos
Organon Motives Inc., Belmont, MA.
mas@organon.com

Organization Domain Modeling (ODM) is a highly tailorable and configurable domain engineering method, useful for diverse organizations and domains, and amenable to integration with a variety of software engineering processes, methods and implementation technologies. The method offers a systematic, exemplar-based approach to analysis of commonality and variability within both legacy systems and requirements for new systems. Under funding by the ARPA STARS Program, ODM has been extensively documented in a guidebook (Version 1.0) [1], as well as in shorter papers [2]. The guidebook provides a formal process model (documented in IDEF-0), workproduct descriptions and templates, and detailed domain engineering guidelines.

ODM is structured in terms of a core domain modeling process, which can be tailored via a set of supporting methods and optional extensions. The core ODM life cycle formalizes an intuitive process that can be applied to a wide variety of domains (including non-software related domains) and on a very small scale (e.g., the entire modeling phase could be prototyped within a few meetings). The method as currently documented, however, is intended to address software-intensive domains within large-scale systems.

ODM reflects the idea that domains are socially defined agreements about an "intended scope of applicability." Domains are always grounded in some "organization context," which might be a company or division, a consortium of multiple organizations or a community of interest such as a standards organization. Stakeholder issues, always a potential problem in any project, turn out to be critical risk factors in domain engineering, which by definition involves designing for multiple contexts of use. The ODM process uses systematic exploration of this stakeholder context to guide selection, scoping and definition of domains strategically aligned with the business interests for the organization(s).

Evolution and Applications. ODM grew out of the design of the Reuse Library Framework (RLF), a STARS-funded technology effort that applied structured inheritance network-based and rule-based knowledge representation techniques to support domain modeling. The method was later refined in close collaboration with Patricia Collins of Hewlett-Packard Company's Software Reuse Initiative, and formed the basis of an HP-proprietary workbook and method used successfully on a number of internal projects [3]. The most extensive application of ODM to date has been on the Army STARS Demonstration Project, in combination with RLF and the Conceptual Framework for Reuse Processes (CFRP) [4,5]. Since completion of the guidebook, several projects (e.g., Rolls-Royce, Logicon) have conducted trial applications almost entirely on the basis of the guidebook alone; this has served as a kind of "stress-testing" for the documentation (reports not yet available). A final guidebook revision (under STARS auspices) is slated for completion in 2Q1996.

Types of Systems Addressed. Some methods assume that domains span the entire scope of a related family of applications, and that a single generic architecture can