A Reuse Reference Grid for Strategic Reuse Goals Assessment

Leslie J. Waguespack
William T. Schiano
Bentley College
lwaguespack@bentley.edu
wschiano@bentley.edu

Abstract

Reuse throughout system life cycles is the most promising organizational policy for cost containment and benefit exploitation available to information system managers today. Large-scale reuse is an expensive endeavor whose benefits are realized when it is applied strategically rather than tactically. The distinction eludes many (and challenges most) IS managers. We present a reuse reference grid for managers to use as an assessment framework to help categorize and assess the cost/benefit of their current level of reuse as a prelude to considering future reuse opportunities.

1. Introduction

Software engineers strive for reduced development and maintenance costs and shortened time to market [8]. The only technological thrust consistently advancing toward these objectives is software reuse [1]. Object oriented technology (OO) and its alter ego, components, have made a dramatic impact on the software development industry, emerging as a key reuse enabler. OO is associated with dramatic successes like that of Brooklyn Union Gas [4] that whet the appetites of managers looking for a competitive edge. Most technology giants have a major reuse effort in place [9]. OO is the backbone technology of e-commerce. Gartner predicted that in 2005, 80 percent of all new application development project spending would employ object-oriented analysis and design [5]. Well-reputed organizations have invested generously in projects using OO and failed in their reuse efforts [6]. Such failed initiatives are regularly criticized for "not having stayed the course long enough" or for "not having adopted the whole paradigm" [7, 11].

This paper explores the concept that it is possible to adopt reuse and supporting OO technologies incrementally and expect incremental benefits. The costs are not mysterious, but the way benefits emerge is somewhat more complex than with previous technologies. We discuss reuse as an economic and organizational goal and propose two dimensions along which organizations choose to position themselves that define their opportunities for reuse and the accompanying costs: the scope of the requirement and sophistication of the abstraction used to address the problem. The requirement scope may vary, from individual application to system and enterprise. The sophistication of the abstraction used to address a problem may range from procedure-driven to data-driven or behavior-driven.

We introduce a reuse reference grid, which combines the dimensions of requirement scope and abstraction sophistication to explore software reuse and the role of OO tools and practice. Managers use the grid to assess the current position of their organization or project. The grid explains the incremental costs and benefits of software reuse when combined with OO or component technology, enabling managers to set goals and targets for reuse programs.

2. A reference model for organizational reuse

The efficacy of reusable components depends on two characteristics: the artifact's requirement scope and the degree to which the abstraction may be specialized or applied elsewhere. On the reuse reference grid (Figure 1) requirement scope is depicted as the horizontal axis and abstraction sophistication as the vertical axis.

0-7695-2507-5/06/$20.00 (C) 2006 IEEE

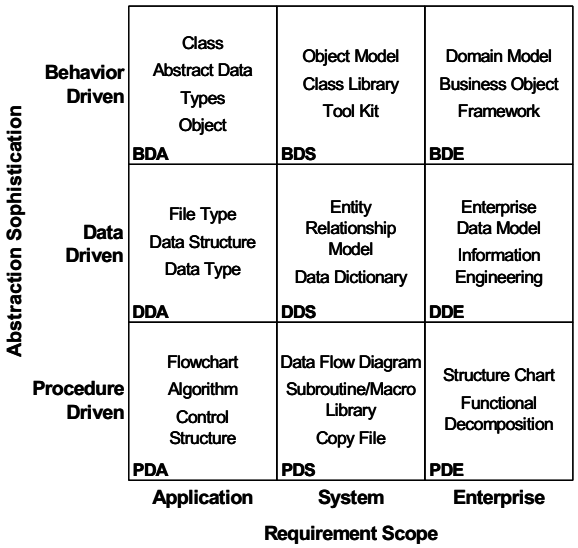

Figure 1
Organizational Reuse Reference Grid

2.1. Requirement scope

Requirement scope bounds what the designer considers in anticipating a component's reuse potential. We define three levels in this dimension:

application: the collection of information attributes and behaviors supporting a business function,

system: the collection of applications and their interrelationships that support a functional area within an enterprise, and

enterprise: the collection of systems that encompass the business information and practices that define the operation of the enterprise or domain as a whole.

Figure 2 illustrates this dimension:

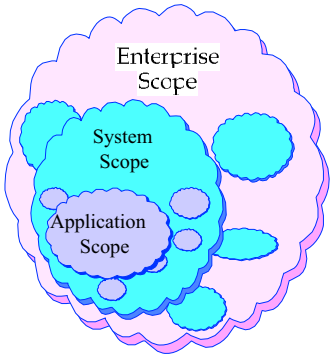

Figure 2
Requirement Scope: Abstraction Applicability

2.2. Abstraction sophistication

We divide abstractions into three groups:

Procedure-driven: "... the principle that any operation that achieves a well-defined effect can be treated by its users as a single entity, despite the fact that the operation may actually be achieved by some sequence of lower-level operations" [3]

Data-driven: "... the principle of defining a data type in terms of the operations that apply to objects of the type, with the constraint that the values of such objects can be modified and observed only by the use of the operations" [3], and

Behavior-driven: the principle of defining patterns of behavior among actors in a situation within an environment along with the stimuli that evoke those behaviors and then discovering the state data and actions that must be present to sustain those patterns of behavior [12].

Procedure-driven modeling is dominated by abstractions that aggregate sequences of modules. Developers can construct applications by combining modules. Functional decomposition is a systems engineering approach grounded in procedural abstraction.

Data-driven modeling using typing, aggregation and association abstracts information and relationships. Developers can construct application systems by combining any data or relationships within the database design. Information engineering is a systems engineering approach based upon data abstraction.

Behavior-driven modeling incorporates aspects of procedure driven and data driven modeling. Through polymorphism and inheritance, OO dialects of behavior driven modeling surpass encapsulation and modularization found in procedure driven modeling, and surpass typing, aggregation and association in data-driven modeling. Object oriented systems engineering and component based systems engineering are based upon behavior driven abstraction.

Procedure-driven modeling focuses on the steps naturally evident in the current practice. It commonly results in a singled-threaded or sequential depiction of problem domain activities. Data-driven modeling would more often focus on the questions that would need answers and commonly results in a collection of un-sequenced queries eventually to be organized in application interface design. Behavior-driven modeling attempts to identify underlying business rules that define *good behavior* in the problem domain. At the same time

it attempts to affix those rules to the tightest focus of responsibility possible. Indeed, both procedure-driven modeling and data-driven modeling suffer from their heritage of sequential-thinking born of the *input-process-output* model of computing implementation. Behavior-driven modeling assumes a more realistic asynchronous interaction of business rules – an approach that does not relegate the consideration of rarely observed business rule combinations to the status of *exceptions.*

3. Reuse economics

Reuse depends on finding an appropriate artifact, understanding its potential for reuse, and applying that artifact in subsequent system building. Finding, understanding and applying all depend on the initial design of the artifact – whether reuse was an intended characteristic during its creation. Reuse is not simply a retrospective activity. It is also a prescriptive activity incorporated in the artifact creation process.

Every newly constructed software system is made up of two types of software: that newly crafted and that previously existing in a form suitable for reuse. We refer to the latter as components. System construction costs result from building from scratch, producing and/or procuring components, and reusing components. If the hardware-reuse experience holds for software, systems built from reusable components will cost less than crafting new software from scratch. The economics of reuse depend on three cost categories. They are:

One-off (OOC) - the cost of developing to satisfy a requirement for a one-time use,

Build-for-reuse (BFRC) - the cost of developing a reusable component to satisfy that same requirement, and

Reuse (RC) - the cost of employing a reusable component(s) to satisfy that same requirement.

Build-for-reuse cost is typically greater than that of One-off cost which is typically greater than that of Reuse, (BFRC > OOC > RC). Build-for-reuse is greater than One-off cost because "reuse" is an added requirement. A reuse requirement affects the domain of analysis (applicability), the design of interfaces (configurability), and the documentation and test packages that accompany these. Reuse is intended to be less than One-off cost specifically because the packaging and preparation of the reusable component obviates that effort in the reuse of it. Reuse cost is the cost of locating, understanding and applying the reusable component.

When deciding whether to build components in-house management must identify requirements that are cost effectively reusable. Management must answer "How reusable can a component be?" and "How many times can a component be reused?" These questions are motivated by basic economics.

The per-reuse cost savings is (OOC minus RC), the one time cost of construction from scratch compared to reusing to accomplish the same functionality. There is a one time added cost of construction for reuse compared to construction from scratch (BFRC minus OOC). If the savings of a single reuse to satisfy any particular requirement (OOC minus RC) were known to be equal or greater than (BFRC minus OOC) then management would always develop Build-for-reuse, components engineered for reuse. However, typically, the per-reuse cost savings is less than the one time added cost and net savings occur only after a component is reused a sufficient number of times to offset the one time added cost. In order to be cost effective, a reuse policy must achieve a Build-for-reuse process that maximizes the opportunity for repeated instances of Reuse and limits the need for One-off. Management's challenge is to determine how much to invest in reuse so that development savings from realized reuse results is a positive return on that investment. In the end, the goal is that systems are developed for less overall cost.

When an organization establishes reuse policy, it is usually referring to the procurement or production of components specifically to manage the relative costs of One-off, Build-for-reuse, and Reuse in its own corporate culture, marketplace and industry. The reuse reference grid described below characterizes factors that influence the prospects for component reuse. Unless these factors are consciously incorporated into the choice to engineer for reuse or not, and how, most reuse is accidental, and unpredictable.

4. Reuse reference grid: costs and benefits

Each cell in the grid illustrates development technologies and practices that characterize an organization's position or an individual project's position on the grid. Each cell implies reuse potential and associated costs of technology adoption and use. While the position of the cells on the grid depicts their relative orientation, they are neither completely discrete nor proportionate. Different projects within the same organization may be positioned in different cells.

The following sections address the character of each of the cells along the two dimensions of reuse and examine what is reused, who reuses it, and how it affects development.

PDA: Procedure driven abstraction in the application scope enables reuse focused on individual application programs. Modelers in this cell use flowcharts,

hierarchical input-process-output diagrams, etc. Structured programming language constructs are readily available, and programmers focus on the description of processing steps to accomplish a program's responsibilities. The individual programmer usually realizes reuse in this cell. The reuse benefits are limited to modifying or extending individual programs. Some well structured and documented program fragments or algorithms may be reused, but because the scope of focus is the individual program, cross-program reuse is difficult and unlikely. If reuse occurs, it usually requires significant adaptation effort to incorporate artifacts beyond the original application. Any reuse realized outside the individual program is more likely accidental than purposeful.

PDS: Procedure driven abstraction in the system scope acknowledges that application function may recur often enough to merit formal definition of reusable program fragments as subroutines or macros. Shared computations and formulae are designed and implemented once, then reused. This type of development for reuse is very common in low level, independently implemented programming such as libraries for numerical functions, execution environment interface, and graphics. Programmers responsible for utility libraries drive this form of reuse. There is little reuse attributable to user level requirements because this cell de-emphasizes the data components of the requirement. Reuse components of this type tend to be standalone and are locally optimized. They are highly sensitive to data formats and structures. Benefits of reuse in this cell lie in the ability to standardize the use of low level system functions, and these components often find their way into higher level programming languages and end-user tools. They are distinguished from the application scope, PDA, by increased emphasis on documentation and cataloging.

PDE: Procedure driven abstraction in the enterprise scope exhibits virtually no extended reuse benefits beyond the system scope. Functional decomposition at the enterprise level does not directly affect system development in any significant reuse capacity. Its primary result is the formulation of system and subsystem boundaries with coupling and cohesion implications at the macro-module level. Source code control systems maximize reuse activities in this cell to manage generations of program modifications. This is more of a cataloging function than an abstraction, improving reuse of existing artifacts, but not promulgating new ones. This activity often introduces a librarian function, but this function is retrospective rather than prospective in regards to user level requirements.

DDA: Data driven abstraction in the application scope is well suited to data intensive problems. Individual programmers are usually trained in application level data modeling and programming techniques. The decision to structure information using a particular data structure (ex. list, graph, binary tree, etc.) usually predetermines the algorithm(s) to be applied – data structure precedes process. With or without data dialects like SQL or QBE, developers may use data typing features in many programming languages to define program specific data abstractions. Choosing and designing appropriate data collections as structures, records or files yield the primary reuse benefits. Application-specific data structures can be reused only in nearly identical applications by copying and adapting their descriptions.

DDS: Data modeling and database management tools characterize data driven abstraction in the system scope. The collection, organization, and retrieval of data can be well accomplished with data meta-languages that treat data as an element of a disciplined data structure. Data languages based on SQL and QBE nearly eliminate the need for procedural programming associated with data storage or retrieval, allowing the developer to describe the characteristics of the output rather than the steps to obtain it. Fourth generation languages, 4GLs, use data languages as their core, along with extended procedural constructs to "facilitate" data manipulation intensive application programming. The benefits of reuse accrue from the data languages themselves, since they hide most of the processing details. Modelers focus on those elements and relationships of data that characterize a business functional area. The stability of data relationships in various transaction activities can be exploited to facilitate reuse, families of reports for example. Indeed, it is considered "good practice" to attempt completeness in describing the data in a domain before beginning individual application development. Reuse revolves around the capabilities enabled by data dictionary functionality. A database engine removed from individual applications manages most stored data access. Titles such as *database administrator* (as opposed to *data administrator*) usually reflect a focus on data required for applications rather than for the business. The stability, accessibility and reliability of the data description are the primary enablers of reuse.

DDE: Data driven abstraction in the enterprise scope relies on developing a single, centralized description of all data in the enterprise domain (sometimes referred to as a data repository). There may be extensive reuse when a centralized description is constantly referenced to define business transactions including collection, verification, summary, and reporting. 4GLs streamline system change when they are capable of incorporating data description changes automatically. Adopting computer aided systems engineering tools, CASE, usually requires developing a thorough and competent

central data description as a first step. Organizations or projects positioned in this cell often employ *data administrators* reflecting a user domain focus. System modeling in this cell must be further augmented if the environment requires complex processing, configuration, or performance aspects. The focus on data abstraction to the virtual exclusion of process issues leaves wide gaps in centrally managed system knowledge. This cell enjoys incremental benefits over those found in the system scope, DDS.

BDA: Behavior driven abstraction in the application scope co-opts the reuse devices of PDA and most of those in DDA. OO programmers use classes to accomplish the functionality of subprograms and data languages (e.g. modules, macros, and abstract data types, and source code generators such as parameterized packages in Ada). The costs of exploiting this technology result from the learning curves of OO programming languages, OOPLs, and OO tools. The expected benefits include greater programmer productivity and increased intra-program reuse. The degree of reuse depends on the emphasis given to reuse by the development team's management. As in this entire column of cells, the reuse is primarily focused on individual programs. But, because OOPLs enable very flexible sub-classing features, accidental reuse occurs more often than with procedure driven or data driven abstraction.

BDS: Behavior driven abstraction in the system scope enables formal reuse management via class library development. A cross-system repository of class definitions screened and tuned for reuse permits significant cost savings in maintenance and modification. The added operational cost consists of screening proposed library artifacts and tuning them for reuse. The management cost entails encouraging application developers to envision their individual efforts as contributions to an overall system object model. The benefits are few if the developers' focus is on individual programs, i.e. they are rewarded for completing programs rather than for contributing to the cross-system asset of reusable classes / components. Reuse repository entries can enjoy extensive reuse when requirements analysis, modeling, and design adopt component reuse as a goal. Organizations positioned in this cell incur greater initial development costs because a reuse adoption effort is an additive cost. Extramural searches for reusable components are an added cost. Benefits in this cell accrue to application development within the local system scope. Such benefits are primarily realized in subsequent development efforts. Analysts, modelers and designers focus on the system scope rather than on individual applications or programs to maximize reuse. Organizations in this scope may employ reuse managers and/or component librarians.

BDE: Behavior driven abstraction in the enterprise scope raises the level of reuse commitment in analysis, modeling and design to that of enterprise-wide consciousness. Description and design decisions affect the opportunity for reuse throughout the enterprise-wide information system scope. In this cell, the philosophy of reuse permeates not only the information system development activities, but the organization's strategic planning. Domain expertise throughout the organization and sometimes the industry is brought to bear. Development resources are dedicated to the search for and exploitation of reuse opportunities across broad expanses of organizational responsibility and activity. Reuse efforts focus on user requirements that collectively define the policies and procedures of the organization. This effort is no less than an enterprise-wide knowledge management activity. Organizations in this cell support enterprise functions titled *reuse engineering* and *domain*/*enterprise modeling*. These organizational activities relate to the highest levels of SEI capability maturity because they involve optimizing the very process of organizational modeling with a goal of enterprise-wide reuse [2]. The BDE cell represents the ultimate undertaking in organizational software reuse.

5. Assessing reuse strategy using the grid

Adopting reuse is not an all or nothing proposition. As the organizational reuse reference grid demonstrates, the range of both the costs and benefits of employing reuse technology vary greatly. The costs and benefits relate as much to the scope of system management that an organization adopts as the technology.

Every systems development effort represents a reuse opportunity. The reuse reference grid facilitates reuse assessment in three ways: 1) categorizing existing reuse, 2) assessing current reuse levels, and 3) considering future reuse strategy.

Software teams use the grid as taxonomy, identifying existing reuse activities and locating each on the grid to identify current reuse. Reuse awareness is critical in developing an organizational strategy. Identifying occurrences of reuse (intended or accidental) is the first step toward managing their costs and benefits.

Once development efforts in a given management scope are positioned in the grid, then commonalities and cross-dependencies can be uncovered that indicate opportunities for system or enterprise-wide reuse, most of which may not have been recognized previously. When managers encourage sharing reuse experiences, they endorse reuse and promote *best practice* within their teams.

Project managers formalize and encourage selected reuse practices in the grid through measurement and behavior reinforcement. Designers assess the potential of extended reuse based upon the current practice; credible expectations grounded in actual experience. System architects exploit the overlap of reuse potential by influencing projects and/or systems, aggregating or partitioning them based upon managed reuse goals.

Initiating and sustaining a reuse program can be quite expensive. Reuse potential must be quite high if a project expects to find cost effectiveness in the upper right of the reuse reference grid. A reuse project team must examine its organization's strategic IS direction. To develop sufficient reuse opportunity, an organization may have to adjust its requirement domain by expanding it to adjacent requirement areas or focusing it to cover more of a particular vertical market. In this case the reuse reference grid is employed as an *outward looking* management device rather than only *inward looking*. Management asks "To what new opportunities in our business or industry-wide requirement domain can we apply our existing reusable resources (or the new ones we are considering)?" Managing the domain of potential reuse is as important as managing the reuse process. To obtain cost effective reuse, there must be sufficient requirement repetition achieved through conscious management of the requirement scope.

In general for an organization or project to pursue reuse more aggressively they must migrate either up or to the right (or both) within the organizational reuse reference grid. As they migrate they incur new costs and meet new reuse opportunities.

5.1. Advancing abstraction sophistication

Upward migration requires deepening a commitment to a modeling abstraction or adopting a new one. Moving from process-driven to data-driven or data-driven to behavior-driven modeling requires acquiring and supporting new skills, methods and tools specific to the modeling abstraction at hand. In large measure the costs and risks of this migration are confined to the IS development activity of the organization and thus this movement is more tactical in nature than strategic.

The maturity of procedure-driven and data-driven technologies makes immersion and/or migration relatively low risk. Many data-driven tools, well-researched and applied theory, and many well-trained and experienced professionals facilitate incorporating the technologies.

Migrating toward behavior-driven abstractions means committing to a less-mature evolving theory, a fast growing and changing inventory of tools, and a relatively young and inexperienced professional workforce. The risk level is higher, if not high, depending on the depth of immersion sought.

5.2 Advancing requirement scope

Migrating an organization to the right within the organizational reuse reference grid indicates a commitment to assessing business functionality from a broader organizational perspective. This increases the opportunity for broad-based component reuse – improving the prospects for cost-justified build-for-reuse activity.

Adopting such a perspective affects project management because of the multiplication of interrelationships that must be understood, documented and modeled. But, perhaps more dramatically, adopting a broader perspective affects organizational strategy because the key to widespread reuse is achieving a clear and well understood organizational direction within which to forecast future information needs and market/domain positioning effectively. The systems under an enterprise mantel of reuse must not only be effectively interfaced, they must be effectively integrated in their contribution to the organizational mission.

The choice to set an organization's reuse policy in the BDE cell is a bold commitment. Adopting reuse engineering as a strategic goal reflects more than a change in system engineering philosophy or the adoption of a modeling paradigm, it is a choice of business model. Sherif and Vinze's [13] found that most barriers to reuse are caused by inadequate efforts by management to support and market reuse.

Telecommunications software manufacturer Sodalia's experience demonstrates how management can overcome such barriers [9]. Organized around a comprehensive reuse commitment, Sodalia exemplifies the integration of OO technology with reuse engineering. Formed in 1993, its charter defines a reuse-focused enterprise to deliver a family of products to a well defined business domain. Not every organization can be born in Sodalia's mold. Evolving an organization toward that mold requires a carefully managed plan; reuse must remain the focus as an organization-wide goal [10].

6. Conclusion

Many organizations have placed great emphasis on languages, tools and technology, but found their reuse experience disappointing; largely, we believe, because they gave insufficient emphasis to requirement scope and failed in their effort to form a clear vision of their reuse cost and benefit goals. The reuse reference grid can be a useful tool helping organizations to clarify their reuse goals and expectations.

Reuse comes in many guises. Strong organizational reuse management commitment using state of the art data driven modeling approaches can achieve potent reuse. For some organizations, this may prove a less costly, lower risk approach using stable, well-understood technologies. If the organizational reuse goals are well formed and the management team is commensurately disciplined, the data driven abstraction approach in the system, DDS, and enterprise, DDE, scopes can yield significant levels of reuse.

Adopting reuse is not an all or nothing proposition. As the reuse reference grid demonstrates, the range of costs and benefits of employing reuse vary greatly. These costs and benefits relate as much to the scope of an organization's requirements management as to the technology. The highest levels of organizational reuse effectiveness found in the BDE cell are tied to the adoption of a behavior driven approach to system modeling and a vision of the business domain as an integrated whole.

7. References

[1] Arsanjani, A. Developing and Integrating Enterprise Components and Services. *Communications of the ACM*, *45* (10), 2002, 31-34.

[2] Bachman, F., Bass, L., Buhman, C., Cornella-Dorda, S., Long, F., Robert, J., Seacord, R. and Wallnau, K. Volume II: Technical Concepts of Component-Based Software Engineering, Carnegie Mellon University, 2000.

[3] Coad, P. and Yourdon, E. *Object Oriented Analysis*. Yourdon Press, Englewood Cliffs, NJ, 1991.

[4] Davis, J. and Morgan, T. Object-Oriented Development at Brooklyn Union Gas. *IEEE Software* (January), 1993, 67-74.

[5] Duggan, J. Successfully Selecting Object-Oriented A&D Tools, Gartner Group, 2002.

[6] Ezran, M., Morisio, M. and Tully, C., Failure and Success Factors in Reuse Programs: A Synthesis of Industrial Experiences,. in *Proceedings of the 1999 International conference on Software engineering*, 1999, 681 - 682.

[7] Fichman, R.G. and Kemerer, C.F. Object Technology and Reuse: Lessons from Early Adopters. *Computer*, *30* (10), 1997, 47-58.

[8] Krueger, C.W. Software Reuse. *ACM Computing Surveys*, *23* (2), 1992, 131-183.

[9] Mambella, E., Ferrari, R., De Carli, F. and Surdo, A.L., An Integrated Approach to Software Reuse Practice. in *Proceedings of the 17th International Conference on Software Engineering on Symposium on Software Reusability*, 1995, 63-80.

[10] Morisio, M., Ezran, M. and Tully, C., Introducing Reuse in Companies: A Survey of European Experiences. in *Proceedings of the 17th International Conference on Software Engineering on Symposium on Software Reusability*, 1999, 63-80.

[11] Pittman, M. Lessons Learned in Managing Object-Oriented Development. *IEEE Software* (January). 1993, 43-53.

[12] Rubin, K.S. and Goldberg, A. Object Behavior Analysis. *Communications of the ACM*, *35* (9), 1992, 48-62.

[13]Sherif, K and Vinze, A., "Barriers to Adoption of Software Reuse: A Qualitative Study," *Information and Management*, 2003, 159-175.