UMIACS–TR–90–47
CS–TR–2446
April 1990

Towards a Comprehensive Framework for Reuse: Model-Based Reuse Characterization Schemes*

Victor R. Basili and H. Dieter Rombach
Institute for Advanced Computer Studies and
Department of Computer Science
University of Maryland
College Park, MD 20742

ABSTRACT

Reuse of products, processes and related knowledge will be the key to enable the software industry to achieve the dramatic improvement in productivity and quality required to satisfy the anticipated growing demands. We need a comprehensive framework of models and model-based characterization schemes for better understanding, evaluating, and planning all aspects of reuse. In this paper we define requirements for comprehensive reuse models and related characterization schemes, assess state-of-the-art reuse characterization schemes relative to these requirements and motivate the need for more comprehensive reuse characterization schemes. We introduce a characterization scheme based upon a general reuse model, apply it and discuss its benefits, and suggest a model for integrating reuse into software development.

PROGRAM LIBRARY
AV WILLIAMS RM. 3164
UNIVERSITY OF MARYLAND

*Research for this study was supported in part by NASA grant NSG-5123, ONR grant NOOO14-87-K-0307 and Airmics grant DE-mail-84OR21400 to the University of Maryland.

TABLE OF CONTENTS:

1 INTRODUCTION 2

2 BASIC REQUIREMENTS FOR A REUSE CHARACTERIZATION SCHEME 3

2.1 Software Development Assumptions 3

2.2 Software Reuse Assumptions 4

2.3 Software Reuse Characteristics 7

3 STATE-OF-THE-ART REUSE CHARACTERIZATION SCHEMES 9

4 MODEL-BASED REUSE CHARACTERIZATION SCHEMES 12

4.1 The Abstract Reuse Model 12

4.2 The First Model Refinement Level 13

4.3 The Second Model Refinement Level 15

4.3.1 Objects-Before-Reuse 15

4.3.2 Objects-After-Reuse 16

4.3.3 Reuse Process 18

5 APPLYING MODEL-BASED REUSE CHARACTERIZATION SCHEMES 20

5.1 Example Reuse Characterizations 20

5.2 Describing/Understanding/Motivating Reuse Scenarios 23

5.3 Evaluating the Cost of Reuse 26

5.4 Planning the Population of Reuse Repositories 27

6 A REUSE-ORIENTED SOFTWARE ENVIRONMENT MODEL 28

7 CONCLUSIONS 31

8 ACKNOWLEDGEMENTS 31

9 REFERENCES 32

1. INTRODUCTION

The existing gap between demand and our ability to produce high quality software cost–effectively calls for an improved software development technology. A reuse oriented development technology can significantly contribute to higher quality and productivity. Quality should improve by reusing proven experience in the form of products, processes and related knowledge such as plans, measurement data and lessons learned. Productivity should increase by using existing experience rather than creating everything from scratch. Many different approaches to reuse have appeared in the literature (e.g., [7, 9, 11, 13, 14, 15, 16, 21, 22, 23]).

Reusing existing experience is a key ingredient to progress in any area. Without reuse everything must be re–learned and re–created; progress in an economical fashion is unlikely. The goal of research in the area of reuse is the achievement of systematic approaches for effectively reusing existing experience to maximize quality and cost benefits.

This paper defines and demonstrates the usefulness of model–based reuse characterization schemes. From a number of important assumptions regarding the nature of software development and reuse we derive four essential requirements for any useful reuse models and related characterization schemes (Section 2). Existing models and characterization schemes are assessed with respect to these assumptions and the need for more comprehensive models and characterization schemes is established (Section 3). We introduce a reuse characterization scheme based on a general model of reuse (Section 4), and discuss its practical application and benefits (Section 5). Throughout the paper we use examples of reusing ***generic Ada packages, design inspections, and cost models*** to demonstrate our approach. Finally, we present a model for integrating and supporting reuse in software development (Section 6).

2. BASIC REQUIREMENTS FOR A REUSE CHARACTERIZATION SCHEME

The reuse approach presented in this paper is based on a number of assumptions regarding software development in general and reuse in particular. These assumptions are based on more than ten years of analyzing software processes and products [1, 3, 4, 5, 6, 19]. This section states our assumptions regarding development and reuse (Sections 2.1 and 2.2, respectively), and derives a set of characteristics required for any useful reuse characterization scheme (Section 2.3).

2.1. Software Development Assumptions

According to a common software development project model depicted in Figure 1, the goal of software development is to produce project deliverables (i.e., project output) that satisfy project needs (i.e., project input) [25]. This goal is achieved according to some development process model which coordinates personnel, practices, methods and tools.

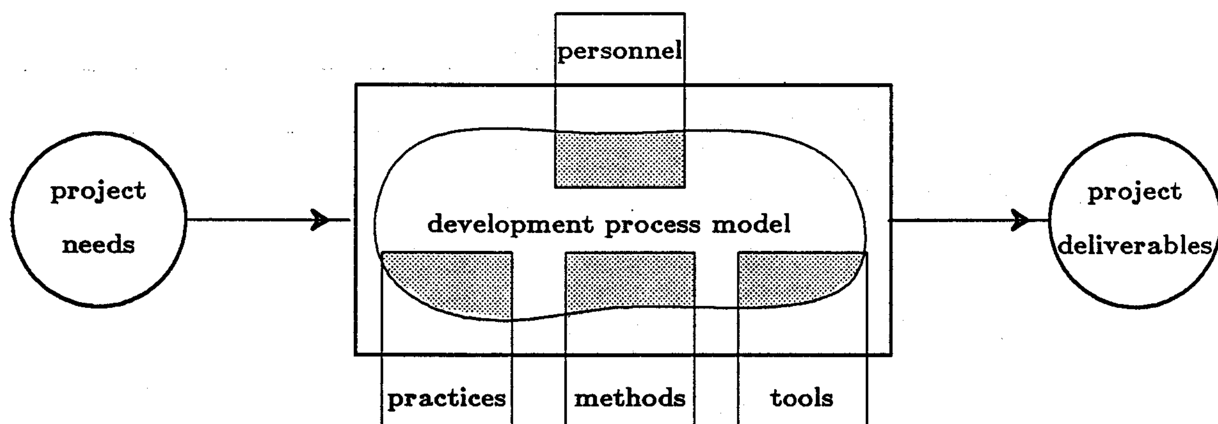

Figure 1: Software Development Project Model

With regard to software development we make the following assumptions:

(D1) A single software development process model cannot be assumed for all software development projects: Different project needs and other project characteristics may suggest and justify different development process models. The potential differences may range from different development process models themselves to different practices, methods and tools supporting these development process models to different personnel.

(D2) Practices, methods and tools – including reuse–related ones – need to be tailored to the project needs and characteristics: Under the assumption that practices, methods and tools support a particular development project, they need to be tailored to the needs and objectives, development process model, and other characteristics of that project.

2.2. Software Reuse Assumptions

Reuse–oriented software development (depicted in Figure 2) assumes that, given the project–specific need to develop an object 'x' that meets specification '$\bar{x}$', we take advantage of some already existing object 'x_k' e {'x_1', ..., 'x_n'} instead of developing 'x' from scratch. In this case, '$\bar{x}$' is not only the specification for 'x' but also the ***reuse specification*** for the set of reuse candidates 'x_1', ..., 'x_n'. Reuse includes the identification of a set of reuse candidates {'x_1', ..., 'x_k', ..., 'x_n'}, the evaluation of their potential to satisfy reuse specification '$\bar{x}$' effectively and the selection of the best–suited candidate 'x_k', the possible modification of the chosen candidate 'x_k' into 'x', and the integration of 'x' into the development process of the current project.

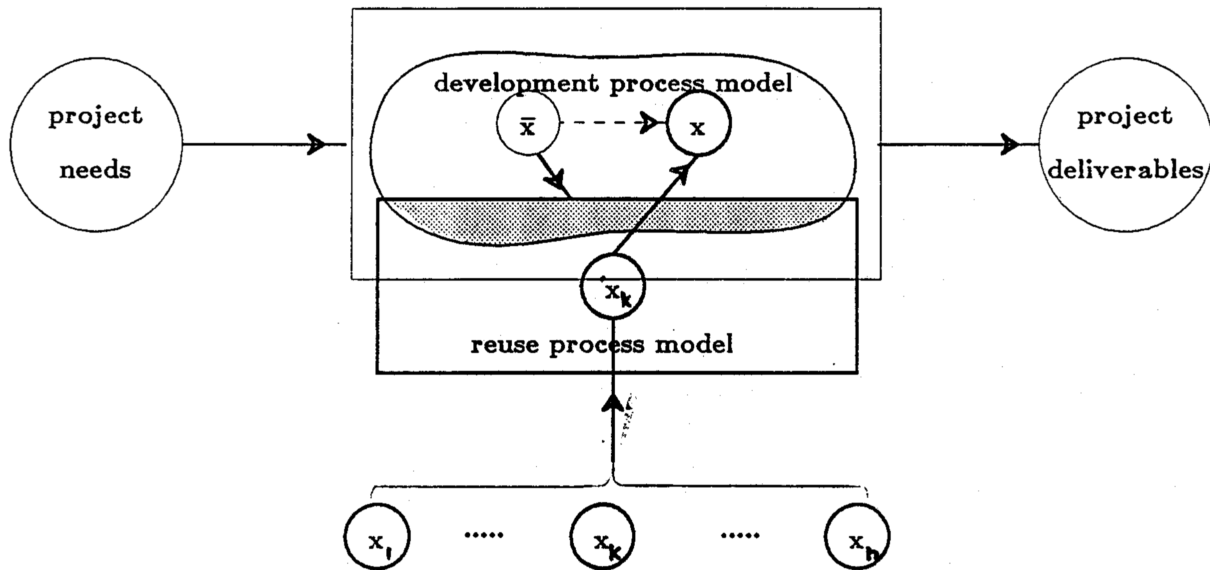

Figure 2: Reuse–Oriented Software Development Model

With regard to software reuse we make the following assumptions:

(R1) All experience can be reused: Typically, the emphasis is on reusing objects of type 'source code'. This limitation reflects the traditional view that software equals code. It ignores the importance of reusing software products across the entire life–cycle (which includes the planning as well as the production phases of a software development project), software processes and methods, and other kinds of knowledge such as models, measurement data or lessons learned.

The reuse of 'generic Ada packages' represents an example of product reuse. Generic Ada packages represent templates for instantiating specific package objects according to a parameter mechanisms. The reuse of 'design inspections' represents an example of process reuse. Design inspections are off–line fault detection and isolation methods applied during the module design phase. They can be based on different techniques for reading (e.g., ad hoc, sequential, control flow oriented, stepwise abstraction oriented). The reuse of 'cost models' represents an example of knowledge reuse. Cost models are used in the estimation, evaluation and control of project cost. They predict cost (e.g., in the form of staff–months) based on a number of characteristic project parameters (e.g., estimated product size in KLoC, product complexity, methodology level).

(R2) Reuse typically requires some modification of the object being reused: Under the assumption that software developments may be different in some way, modification of

experience from prior projects must be anticipated. The degree of modification depends on how many, and to what degree, existing object characteristics differ from their desired characteristics.

To reuse an Ada package 'list of integers' to organize a 'list of reals' we need to modify it. We can either modify the existing package by hand, or we can use a generic package 'list' which can be instantiated via a parameter mechanism for any base type.

To reuse a design inspection method across projects characterized by significantly different fault profiles, the underlying reading technique may need to be tailored to the respective fault profiles. If 'interface faults' replace 'control flow faults' as the most common fault type, we can either select a different reading technique all together (e.g., step-wise abstraction instead of control-flow oriented) or we can establish specific guidelines for identifying interface faults.

To reuse a cost model across projects characterized by different application domains, we may have to change the number and type of characteristic project parameters used for estimating cost as well as their impact on cost. If 'commercial software' is developed instead of 'real-time software', we may have to consider re-defining 'estimated product size' to be measured in terms of 'data structures' instead of 'lines of code' or re-computing the impact of the existing parameters on cost. Using a cost model effectively implies a constant updating of our understanding of the relationship between project parameters and cost.

(R3) Analysis is necessary to determine when and if reuse is appropriate: The decision to reuse existing experience as well as how and when to reuse it needs to be based on an analysis of the payoff. Reuse payoff is not always easy to evaluate. We need to understand (i) the objectives of reuse, (ii) how well the available reuse candidates are qualified to meet these objectives, and (iii) the mechanisms available to perform the necessary modification.

Assume the existence of a set of Ada generics which represent application-specific components of a satellite control system. The objective may be to reuse such components to build a new satellite control system of a similar type, but with higher precision. Whether the existing generics are suitable depends on a variety of characteristics: Their correctness and reliability, their performance in prior instances of reuse, their ease of integration into a new system, the potential for achieving the higher degree of precision through instantiation, the degree of change needed, and the existence of reuse mechanisms that support this change process. Candidate Ada generics may theoretically be well suited for reuse; however, without knowing the answers to these questions, they may not be reused due to lack of confidence that reuse will pay off.

Assume the existence of a design inspection method based on ad-hoc reading which has been used successfully on past satellite control software developments within a standard waterfall model. The objective may be to reuse the method in the context of the Cleanroom development method [18, 20]. In this case, the method needs to be applied in the context of a different life-cycle model, different design approach, and different design representations. Whether and how the existing method can be reused depends on our ability to tailor the reading technique to the stepwise refinement oriented design technique used in Cleanroom, and the required intensity of

reading due to the omission of developer testing. This results in the definition of the stepwise abstraction oriented reading technique [8].

Assume the existence of a cost model that has been validated for the development of satellite control software based on a waterfall life-cycle model, functional decomposition oriented design techniques, and functional and structural testing. The objective may be to reuse the model in the context of Cleanroom development. Whether the cost model can be reused at all, how it needs to be calibrated, or whether a completely different model may be more appropriate depends on whether the model contains the appropriate variables needed for the prediction of cost change or whether they simply need to be re-calibrated. This question can only be answered through thorough analysis of a number of Cleanroom projects.

(R4) Reuse must be integrated into the specific software development: Reuse is intended to make software development more effective. In order to achieve this objective we need to tailor reuse practices, methods and tools towards the respective development process.

We have to decide when and how to identify, modify and integrate existing Ada packages. If we assume identification of Ada generics by name, and modification by the generic parameter mechanism, we require a repository consisting of Ada generics together with a description of the instantiation parameters. If we assume identification by specification, and modification of the generic's code by hand, we require a suitable specification of each generic, a definition of semantic closeness of specifications so we can find suitable reuse candidates, and the appropriate source code documentation to allow for ease of modification. In the case of identification by specification we may consider identifying reuse candidates at high-level design (i.e., when the component specifications for the new product exist) or even when defining the requirements.

We have to decide on how often, when, and how design inspections should be integrated into the development process. If we assume a waterfall-based development life-cycle, we need to determine how many design inspections need to be performed and when (e.g., once for all modules at the end of module design, once for all modules of a subsystem, or once for each module). We need to state which documents are required as input to the design inspection, what results are to be produced, what actions are to be taken, and when, in case the results are insufficient, and who is supposed to participate.

We have to decide when to initially estimate cost and when to update the initial estimate. If we assume a waterfall-based development life-cycle, we may estimate cost initially based on estimated product and process parameters (e.g., estimated product size). After each milestone, the estimated cost can be compared with the actual cost. Possible deviations are used to correct the estimate for the remainder of the project.

2.3. Software Reuse Characteristics

The above software reuse assumptions suggest that 'reuse' is a complex concept. We need to build models and characterization schemes that allow us to define and understand, compare and evaluate, and plan the objectives of reuse, the candidate objects of reuse, the reuse process itself,

and the potential for effective reuse. Based upon the above assumptions, such models and characterization schemes need to exhibit the following characteristics:

(C1) **Applicable to all types of reuse objects**: We want to be able to characterize products, processes and all other types of related knowledge such as plans, measurement data or lessons learned.

(C2) **Capable of characterizing objects–before–reuse and objects–after–reuse**: We want to be able to characterize the reuse candidates (from here on called 'objects–before–reuse') as well as the object actually being reused in the current project (from here on called 'object–after–reuse'). This will enable us to (i) judge the suitability of a given reuse candidate based on the distance between its actual before–reuse and desired after–reuse characteristics, and (ii) establish criteria for useful reuse candidates (object–before–reuse characteristics) based on anticipated objectives for their (re)use (object–after–reuse characteristics).

(C3) **Capable of characterizing the reuse process itself**: We want to be able to (i) judge the ease of bridging the gap between different object characteristics before– and after–reuse, and (ii) derive additional criteria for useful reuse candidates based on characteristics of the reuse process itself.

(C4) **Capable of being systematically tailored to specific project (i.e., development and reuse) needs and other characteristics**: We want to be able to adjust a given reuse characterization scheme to changing needs in a systematic way. This requires not only the ability to change the scheme, but also some kind of rationale that ties the given reuse characterization scheme back to its underlying model and assumptions. Such a rationale enables us to identify the impact of different environments and modify the scheme in a systematic way.

3. STATE–OF–THE–ART REUSE CHARACTERIZATION SCHEMES

A number of research groups have developed characterization schemes for reuse (e.g., [9, 11, 13, 21, 22]). The schemes can be distinguished as *special purpose schemes* and *meta schemes*.

The large majority of published characterization schemes have been developed for a special purpose. They consist of a fixed number of characterization dimensions. There intention is to characterize software products as they exist. Typical dimensions for characterizing source code objects in a repository are "function", "size", or "type of problem". Examples schemes include the schemes published in [11, 13], the ACM Computing Reviews Scheme, AFIPS's Taxonomy of Computer Science and Engineering, schemes for functional collections (e.g., GAMS, SHARE, SSP, SPSS, IMSL) and schemes for commercial software catalogs (e.g., ICP, IDS, IBM Software Catalog, Apple Book). It is obvious that special purpose schemes are not designed to satisfy the reuse modeling characteristics of section 2.3.

A few characterization schemes can be instantiated for different purposes. They explicitly acknowledge the need for different schemes (or the expansion of existing ones) due to different or changing needs of an organization. They, therefore, allow the instantiation of any imaginable scheme. An excellent example is Ruben Prieto–Diaz's facet–based meta–characterization scheme [14, 17]. Theoretically, meta schemes are flexible enough to allow the capturing of any reuse aspect. However, based on known examples of actual uses of meta schemes, such broadness seems not intended. Instead, most examples focus on product reuse, are limited to the objects–before–reuse, and ignore the reuse process entirely. Meta schemes were also not designed to satisfy the reuse modeling characteristics of section 2.3.

We have found that existing schemes – special purpose as well as meta schemes – do not satisfy our requirements. To illustrate the problems associated with their limitations, we use the following example scheme which can be viewed either as a special–purpose scheme or a specific

instantiation of a meta scheme*:

Each reuse candidate is characterized in terms of

- **name**: What is the object's name? (e.g., buffer.ada, sel_inspection, sel_cost_model)
- **function**: What is the functional specification or purpose of the object? (e.g., integer_queue, <element>_buffer, sensor control system, certify appropriateness of design documents, predict project cost)
- **use**: How can the object be used? (e.g., product, process, knowledge)
- **type**: What type of object is it? (e.g., requirements document, code document, inspection method, coding method, specification tool, graphic tool, process model, cost model)
- **granularity**: What is the object's scope? (e.g., system level, subsystem level, component level, module – package, procedure, function – level, entire life cycle, design stage, coding stage)
- **representation**: How is the object represented? (e.g., data, informal set of guidelines, schematized templates, formal mathematical model, languages such as Ada, automated tools)
- **input/output**: What are the external input/output dependencies of the object needed to completely define/extract it as a self–contained entity? (e.g., global data referenced by a code unit, formal and actual input/output parameters of a procedure, instantiation parameters of a generic Ada package, specification and design documents needed to perform a design inspection, defect data produced by a design inspection, variables of a cost model)
- **dependencies**: What are additional assumptions and dependencies needed to understand the object? (e.g., assumption on user's qualification such as knowledge of Ada or qualification to read, specification document to understand a code unit, readability of design document, homogeneity of problem classes and environments underlying a cost model)
- **application domain**: What application classes was the object developed for? (e.g. ground support software for satellites, business software for banking, payroll software)
- **solution domain**: What environment classes was the object developed in? (e.g., waterfall life–cycle model, spiral life–cycle model, iterative enhancement life–cycle model, functional decomposition design method, standard set of methods)
- **object quality**: What qualities does the object exhibit? (e.g., level of reliability, correctness, user–friendliness, defect detection rate, predictability)

Let's assess the above reuse characterization scheme relative to the four desired characteristics of section 2.3:

(C1) It is theoretically possible to characterize all types of experience according to the above scheme (in case of a meta scheme we could even create new ones). For example, a generic Ada package 'buffer.ada' may be characterized as having identifier 'buffer.ada', offering the function '<element>_buffer', being usable as a 'product' of type 'code document' at the 'package module level', and being represented in 'Ada'. The self–contained definition of the package requires knowledge regarding the instantiation parameters as well as its visibility of externally

* Characterization dimensions are marked with '–'; example categories for each dimension are listed in parenthesis.

defined objects (e.g., explicit access through WITH clauses, implicit access according to nesting structure). In addition, effective use of the object may require some basic knowledge of the language Ada and assume thorough documentation of the object itself. It may have been developed within the application domain 'ground support software', according to a 'waterfall life–cycle' and 'functional decomposition design', and exhibiting high quality in terms of 'reliability'.

(C2) The scheme is used to characterize reuse candidates (i.e., objects–before–reuse) only. However, in order to evaluate the reuse potential of an object–before–reuse in a given reuse scenario, one needs to understand the distance between its characteristics and the characteristics of the desired object (i.e., object–after–reuse). In the case of the Ada package example, the required function may be different, the quality requirements with respect to reliability may be higher, or the design method used in the current project may be different from the one according to which the package has been created originally. Without understanding the distance to be bridged between reuse requirements and reuse candidates it is hard to (a) predict the cost involved in reusing a particular object, and (b) establish criteria for populating a reuse repository that supports cost–effective reuse.

(C3) The scheme is not intended to characterize the reuse process at all. To really predict the cost of reuse we do not only have to understand the distance to be bridged between objects–before and objects–after–reuse (as pointed out above), but also the intended process to bridge it (i.e., the reuse process). For example, it can be expected that it is easier to bridge the distance with respect to function by using a parameterized instantiation mechanism rather than modifying the existing package by hand.

(C4) Their is no explicit rationale for the eleven dimensions of the example scheme. That makes it hard to reason about its appropriateness as well as modify it in any systematic way. There is no guidance in tailoring the example scheme to new needs neither with respect to what is to changed (e.g., only some categories, dimensions, or the entire implicitly underlying model) nor

how it is to be changed.

The result of this assessment suggests the urgent need for new, better reuse characterization schemes. In the next section, we suggest a model–based scheme which satisfies all four characteristics.

4. MODEL–BASED REUSE CHARACTERIZATION SCHEMES

In this section we define a model–based reuse characterization scheme satisfying the characteristics (C1–4) stated in section 2.3. We start this modeling approach with a very general reuse model satisfying satisfying the reuse assumptions, refine it step by step until it generates reuse characterization dimensions at the level of detail needed to understand, evaluate, motivate or improve reuse. This modeling approach allows us to deal with the complexity of the modeling task itself, and document an explicit rationale for the resulting model.

4.1. The Abstract Reuse Model

The general reuse model used in this section is consistent with the view of reuse represented in section 2.2. It assumes the existence of objects–before–reuse and objects–after–reuse, and a transformation between the two:

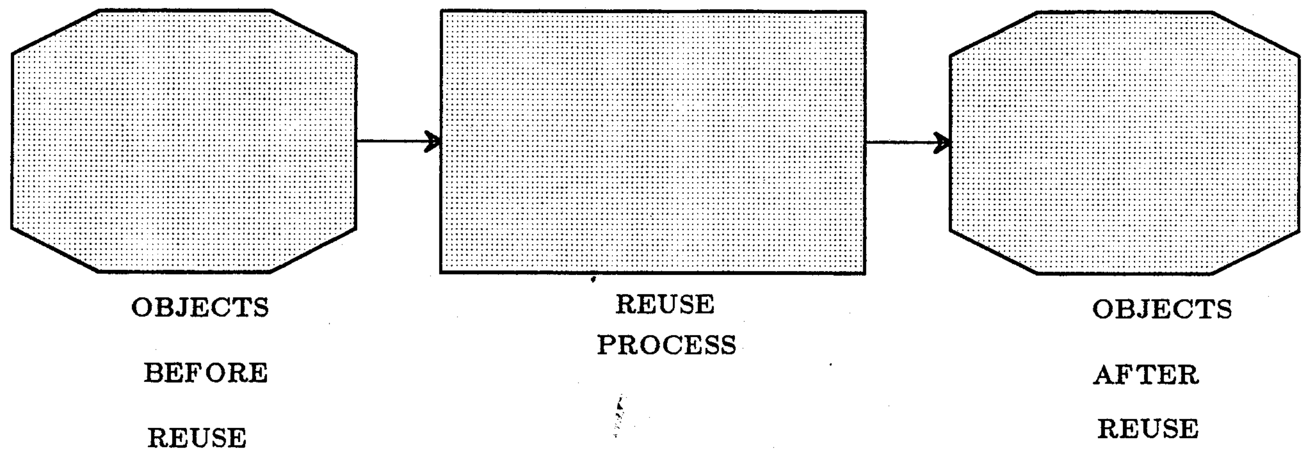

Figure 3: Abstract Reuse Model (Refinement level 0)

The objects–before–reuse represent experience from prior projects, have been evaluated as being of potential reuse value, and have been made available in some form of a repository. The objects–after–reuse are the (potentially modified) versions of objects–before–reuse integrated into some project other than the one they were initially created for. Object–after–reuse characteristics represent the 'reuse specification' for any candidate 'object–before–reuse'. Both the objects–before–reuse and the objects–after–reuse may represent any type of experience accumulated in the context of software projects ranging from products to processes to knowledge. The reuse process transforms objects–before–reuse into objects–after–reuse.

4.2. The First Model Refinement Level

Figure 4 depicts the result of the first refinement step of the general model of Figure 3.

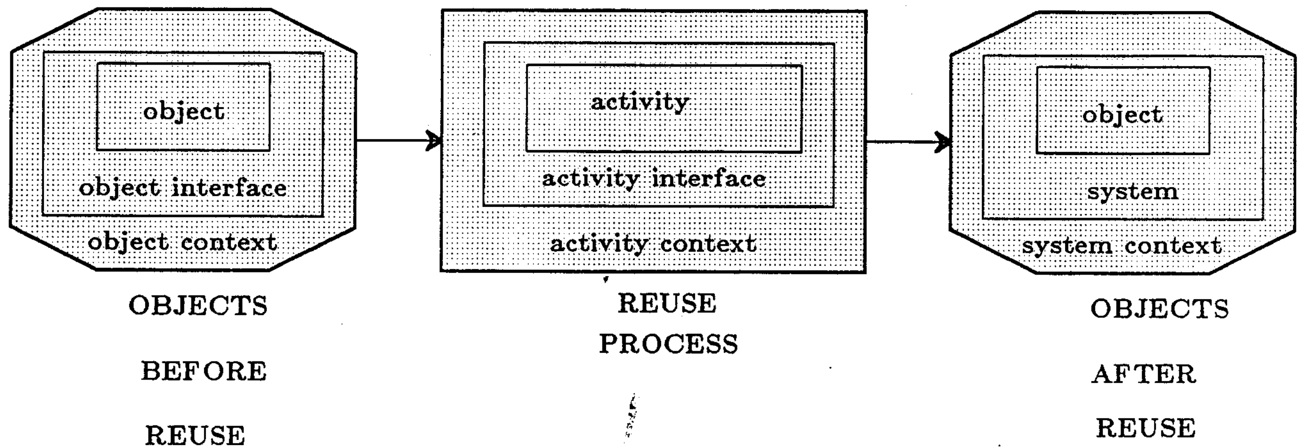

Figure 4: Our Reuse Model (Refinement level 1)

Each *object–before–reuse* is a specific candidate for reuse. It has various attributes that describe and bound the object. Most objects are physically part of a system, i.e. they interact with other objects to create some greater object. If we want to reuse an object we must understand its interaction with other objects in the system in order to extract it as a unit, i.e. *object interface*. Objects were created in some environment which leaves its characteristics on the object, even though those characteristics may not be visible. We call this the *object context*.

The *object–after–reuse* is a specification for a set of before–reuse candidates. Therefore, we may have to consider different attributes. The *system* in which the transformed object is integrated and the *system context* in which the system is developed must also be classified.

The *reuse process* is aimed at extracting the object–before–reuse from a repository based on the available object–after–reuse characteristics, and making it ready for reuse in the system and context in which it will be reused. We must describe the various *reuse activities* and classify them. The reuse activities need to be integrated into the reuse–enabling software development process. The means of integration constitute the *activity interface*. Reuse requires the transfer of experience across project boundaries. The organizational support provided for this experience transfer is referred to as *activity context*.

Based upon the goals for the specific project, as well as the organization, we must evaluate (i) the required qualities of the object–after–reuse, (ii) the quality of the reuse process, especially its integration into the enabling software evolution process, and (iii) the quality of the existing objects–before–reuse.

4.3. The Second Model Refinement Level

Each component of the First Model Refinement (Figure 4) is further refined as depicted in Figures 5(a–c) . It needs to be noted that these refinements are based on our current understanding of reuse and may, therefore, change in the future.

4.3.1. Objects–Before–Reuse

In order to characterize the object itself, we have chosen to provide the following six dimensions and supplementing categories: the object's name (e.g., buffer.ada), its function (e.g., integer_buffer), its possible use (e.g., product), its type (e.g., requirements document), its granularity (e.g., module), and its representation (e.g., Ada language). The object interface consists of such things as what are the explicit inputs/outputs needed to define and extract the object as a self–contained unit (e.g., instantiation parameters in the case of a generic Ada package), and what are additionally required assumptions and dependencies (e.g., user's knowledge of Ada). Whereas the object and object interface dimensions provide us with a snapshot of the object at hand, the object context dimension provides us with historical information such as the application classes the object was developed for (e.g., ground support software for satellites), the environment the object was developed in (e.g., waterfall life–cycle model), and its validated or anticipated quality (e.g., reliability).

The resulting model refinement is depicted in Figure 5a.

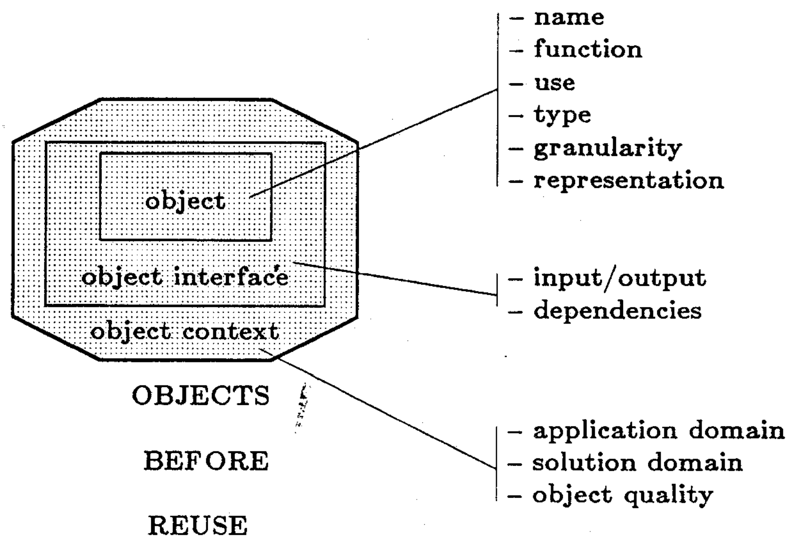

Figure 5a: Reuse Model (Objects–Before–Reuse / Refinement level 2)

A detailed definition of the above eleven dimensions – together with example categories – has already been presented in Section 3. In contrast to Section 3, we now have (i) a rationale for these dimensions (see Figure 5a) and (ii) understand that they cover only part (i.e., the objects–before–reuse) of the comprehensive reuse model depicted in Figure 4.

4.3.2. Objects–After–Reuse

In order to characterize objects–after–reuse, we have chosen the same eleven dimensions and supporting categories as for the objects–before–reuse. The resulting model refinement is depicted in Figure 5b:

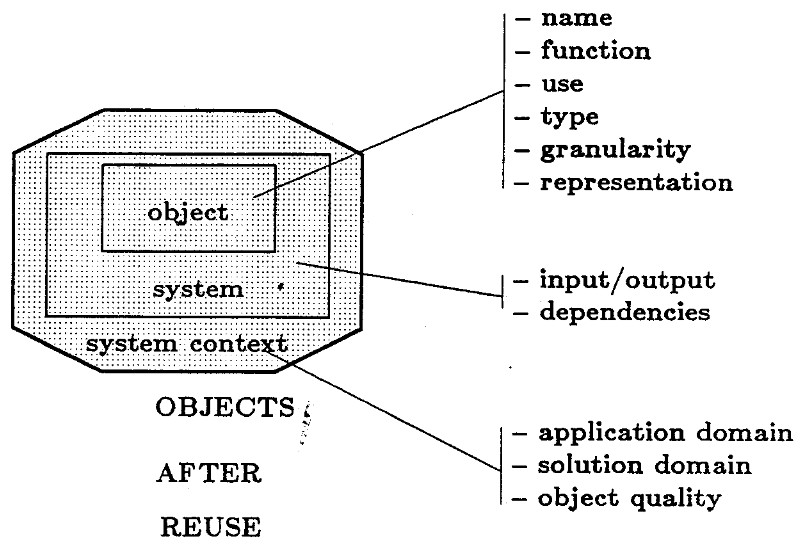

Figure 5b: Reuse Model (Objects–After–Reuse / Refinement level 2)

However, an object may change its characteristics during the actual process of reuse. Therefore, its characterizations before–reuse and after–reuse can be expected to be different. For example, an object–before–reuse may be a compiler (type) product (use), and may have been developed according to a waterfall life–cycle approach (solution domain). The object–after–reuse may be a compiler (type) process (use) integrated into a project based on iterative enhancement (solution domain).

This means that despite the similarity between the refined models of objects–before–reuse and objects–after–reuse, there exists a significant difference in emphasis: In the former case the emphasis is on the potentially reusable objects themselves; in the latter case, the emphasis is on the system in which these object(s) are (or are expected to be) reused. This explains the use of different dimension names: 'system' and 'system context' instead of 'object interface' and 'object context'.

The distance between the characteristics of an object–before–reuse and an object–after–reuse give an indication of the gap to be bridged in the event of reuse.

4.3.3. Reuse Process

The reuse process consists of several activities. In the remainder of this paper, we will use a model consisting of four basic activities: identification, evaluation, modification, and integration. In order to characterize each reuse activity we may be interested in its name (e.g., modify.p1), its function (e.g., modify an identified reuse candidate to entirely satisfy given object–after–reuse characteristics), its type (e.g., modification), and the mechanism used to perform its function (e.g., modification via parameterization). The interface of each activity may consist of such things as what the explicit input/output interfaces between the activity and the enabling software evolution environment are (e.g., in the case of modification: performed during the coding phase, assumes the existence of a specification), and what other assumptions regarding the evolution environment need to be satisfied (e.g., existence of certain configuration control policies). The activity context may include information about how experience is transferred from the object–before–reuse domain to the object–after–reuse domain (experience transfer), and the quality of each reuse activity (e.g., reliability, productivity).

This refinement of the reuse process is depicted in Figure 5c.

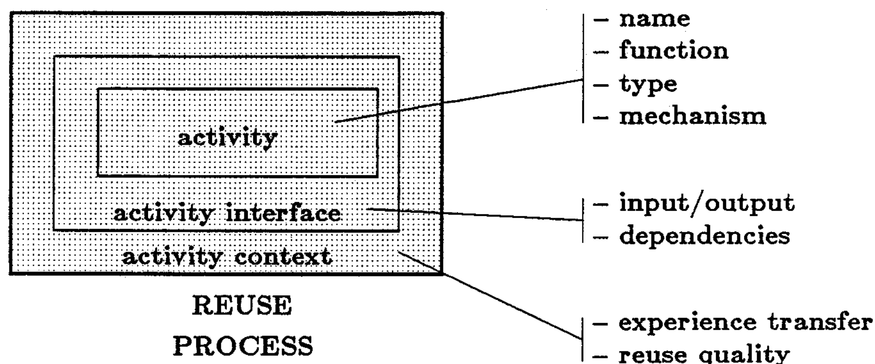

Figure 5c: Reuse Model (Reuse Process / Refinement level 2)

In more detail, the dimensions and example categories for characterizing the reuse process are:

- **REUSE PROCESS**: For each reuse activity characterize:
  + **Activity**:
    - **name**: What is the name of the activity? (e.g., identify.generics, evaluate.generics, modify.generics, integrate.generics)
    - **function**: What is the function performed by the activity? (e.g., select candidate objects $\{x_i\}$ which satisfy certain object categories of the object–after–reuse specification '$\bar{x}$'; evaluate the potential of the selected candidate objects of satisfying the given system and system context dimensions of the object–after–reuse specification '$\bar{x}$' and pick the most suited candidate 'x_k'; modify 'x_k' to entirely satisfy '$\bar{x}$'; integrate object 'x' into the current development project)
    - **type**: What is the type of the activity? (e.g., identification, evaluation, modification, integration)
    - **mechanism**: How is the activity performed? (in the case of identification: e.g., by name, by function, by type and function; in the case of evaluation: e.g., by subjective judgement, by evaluation of historical baseline measurement data; in the case of modification: e.g., verbatim, parameterized, template–based, unconstrained; in the case of integration: e.g., according to the system configuration plan, according to the project/process plan)
  + **Activity Interface**:
    - **input/output**: What are explicit input and output interfaces between the reuse activity and the enabling software evolution environment? (in the case of identification: e.g., specification for the needed object–after–reuse / set of candidate objects–before–reuse; in the case of modification: e.g., one selected object–before–reuse, specification for the needed object–after–reuse / object–after–reuse)
    - **dependencies**: What are other implicit assumptions and dependencies on data and information regarding the software evolution environment? (e.g., time at which reuse activity is performed – relative to the enabling development process: e.g., during design or coding stages; additional information needed to perform the reuse activity effectively: e.g., package specification to instantiate a generic package, knowledge of system configuration plan, configuration management procedures, or project plan)
  + **Activity Context**:
    - **experience transfer**: What are the support mechanisms for transferring experience across projects? (e.g., human, experience base, automated)
    - **reuse quality**: What is the quality of each reuse activity? (e.g., high reliability, high predictability of modification cost, correctness, average performance)

5. APPLYING MODEL–BASED REUSE CHARACTERIZATION SCHEMES

We demonstrate the applicability of our model–based reuse scheme by characterizing three hypothetical reuse scenarios related to product, process and knowledge reuse: Ada generics, design inspections, and cost models (Section 5.1). The characterization of the Ada generics scenario is furthermore used to demonstrate the benefits of model–based characterizations to describe/understand/motivate a given reuse scenario (Section 5.2), to evaluate the cost of reuse (Section 5.3), and to plan the population of a reuse repository (Section 5.4).

5.1. Example Reuse Characterizations

The characterization scheme of section 4 has been applied to the three examples of product, process and knowledge reuse introduced in section 2. The resulting characterizations are contained in tables 2, 3, and 4:

Dimensions	Reuse Examples: Ada generic	design inspection	cost model
name	buffer.ada	sel_inspection.waterfall	sel_cost_model.fortran
function	<element>_buffer	certify appropriateness of design documents	predict project cost
use	product	process	knowledge
type	code document,	inspection method	cost model
granularity	package	design stage	entire life cycle
representation	Ada/ generic package	informal set of guidelines	formal mathematical model
input/output	formal and actual instantiation params	specification and design document needed, defect data produced	estimated product size in KLOC, complexity rating, methodology level, cost in staff_hours
dependencies	assumes Ada knowledge	assumes a readable design, qualified reader	assumes a relatively homogeneous class of problems and environments
application domain	ground support sw for satellites	ground support sw for satellites	ground support sw for satellites
solution domain	waterfall (Fortran) life-cycle model, functional de-composition design method	waterfall (Fortran) life-cycle model, standard set of methods	waterfall (Fortran) life-cycle model standard set of methods
object quality	high reliability (e.g., < 0.1 defects per KLoC for a given set of acceptance tests)	average defect detection rate (e.g., > 0.5 defects detected per staff_hour)	average predictability (e.g., < 5% prediction error)

Table 2: Characterization of Example Reuse Objects–Before–Reuse

Dimensions	Reuse Examples: Ada generics	design inspection	cost model
name	string_buffer.ada	sel_inspection.cleanroom	sel_cost_model.ada
function	string_buffer	certify appropriateness of design documents	predict project cost
use	product	process	knowledge
type	code document,	inspection method	cost model
granularity	package	design stage	entire life cycle
representation	Ada	informal set of guidelines	formal mathematical model
input/output	formal and actual instantiation params	specification and design document needed, defect data produced	estimated product size in KLOC, complexity rating, methodology level, cost in staff_hours
dependencies	assumes Ada knowledge	assumes a readable design, qualified reader	assumes a relatively homogeneous class of problems and environments
application domain	ground support sw for satellites	ground support sw for satellites	ground support sw for satellites
solution domain	waterfall (Ada) life-cycle model, object oriented design method	Cleanroom (Fortran) development model, stepwise refinement oriented design, statistical testing	waterfall (Ada) life-cycle model, revised set of methods
object quality	high reliability (e.g., $<$ 0.1 defects per KLoC for a given set of acceptance tests), high performance (e.g., max. response times for a set of tests)	high defect detection rate (e.g., $>$ 1.0 defects detected per staff_hour) wrt. interface faults	high predictability (e.g., $<$ 2% prediction error)

Table 3: Characterization of Example Reuse Objects–After–Reuse

Dimensions	Reuse Examples		
	Ada generics	design inspection	cost model
name	modify.generics	modify.inspections	modify.cost_models
function	modify to satisfy target specification	modify to satisfy target specification	modify to satisfy target specification
type	modification	modification	modification
mechanism	parameterized (generic mechanism)	unconstrained	template-based
input/output	buffer.ada, reuse specification/ string_buffer.ada	sel_inspection.waterfall, reuse specification/ sel_inspection.cleanroom	sel_cost_model.fortran, reuse specification/ sel_cost_model.ada
dependencies	performed during coding stage, package specification needed, knowledge of system configuration plan	performed during planning stage, knowledge of project plan	performed during planning stage, knowledge of historical project profiles
experience transfer	experience base	human and experience base	human and experience base
reuse quality	correctness	correctness	correctness

Table 4: Characterization of Example Reuse Processes

5.2. Describing/Understanding/Motivating Reuse Scenarios

We will demonstrate the benefits of our reuse characterization scheme to describe, understand, and motivate the reuse of Ada generics as characterized in section 5.1.

We assume that in some project the need has arisen to have an Ada package implementing a 'string_buffer' with high 'reliability and performance' characteristics. This need may have been established during the project planning phase based on domain analysis, or during the design or coding stages. This package will be integrated into a software system designed according to

object–oriented principles. The complete reuse specification is contained in Table 3.

First, we identify candidate objects based on some subset of the object related characteristics stated in Table 3: string_buffer.ada, string_buffer, product, code document, package, Ada. The more characteristics we use for identification, the smaller the resulting set of candidate objects will be. For example, if we include the name itself, we will either find exactly one object or none. Identification may take place during any project stage. We will assume that the set of successfully identified reuse candidates contains 'buffer.ada', the object characterized in Table 2.

Now we need to evaluate whether and to what degree 'buffer.ada' (as well as any other identified candidate) needs to be modified and estimate the cost of such modification compared to the cost required for creating the desired object 'string_buffer' from scratch. Three characteristics of the chosen reuse candidate deviate from the expected ones: it is more general than needed (see function dimension), it has been developed according to a different design approach (see solution domain dimension), and it does not contain any information about its performance behavior (see object quality dimension). The functional discrepancy requires instantiating object 'buffer.ada' for data type 'string'. The cost of this modification is extremely low due to the fact that the generic instantiation mechanism in Ada can be used for modification (see Table 4). The remaining two discrepancies cannot be evaluated based on the information available through the characterizations in section 5.1. On the one hand, ignoring the solution domain discrepancy may result in problems during the integration phase. On the other hand, it may be hard to predict the cost of transforming 'buffer.ada' to adhere to object–oriented principles. Without additional information about either the integration of non–object–oriented packages or the cost of modification, we only have the choice between two risks. Predicting the cost of changes necessary to satisfy the stated object performance requirements is impossible because we have no information about the candidate's performance behavior. It is noteworthy that very often practical reuse seems to fail because of lack of appropriate information to evaluate the reuse implications a–priori, rather than because of technical infeasibility.

In case the object characterized in Table 2 has been modified successfully to satisfy the specification in Table 3, we need to integrate it into the ongoing development process. This task needs to be performed consistently with the system configuration plan and the process plan used in this project.

The characterization of both objects (before/after–reuse) and the reuse process allow us to understand some of the implications and risks associated with discrepancies between identified reuse candidates and target reuse specification. Problems arise when we have either insufficient information about the existence of a discrepancy (e.g., object performance quality in our example), or no understanding of the implications of an identified discrepancy (e.g., solution domain in our example). In order to avoid the first type of problem, one may either constrain the identification process further by including characteristics other than just the object related ones, or not have any objects without 'performance' data in the reuse repository. If we had included 'desired solution domain' and 'object performance' as additional criteria in our identification process, we may not have selected object 'buffer.ada' at all. If every object in our repository would have performance data attached to it, we at least would be able to establish the fact that there exists a discrepancy. In order to avoid the second type of problem, we need have some (semi–) automated modification mechanism, or at least historical data about the cost involved in similar past situations. It is clear that in our example any functional discrepancy within the scope of the instantiation parameters is easy to bridge due to the availability of a completely automated modification mechanism (i.e., generic instantiation in Ada). Any functional discrepancy that cannot be bridged through this mechanisms poses a larger and possibly unpredictable risk. Whether it is more costly to re–design 'buffer.ada' in order to adhere to object oriented design principles or to re–develop it from scratch is not obvious without past experience.

Based on the preceding discussion, the motivational benefits are that we have a sound rationale for suggesting the use of certain reuse mechanisms (e.g., automated in the case of Ada packages to reduce the modification cost), criteria for populating a reuse repository (e.g., do

exclude objects without performance data to avoid the unnecessary expansion of the search space), criteria for identifying reuse candidates effectively according to some reuse specification (e.g., do include solution domain to avoid the identification of candidates with unpredictable modification cost), or certain types of reuse specifications (e.g., require that each reuse request is specified in terms of all object dimensions, except probably name, and all system context dimensions).

5.3. Evaluating the Cost of Reuse

We will demonstrate the benefits of our reuse characterization scheme to evaluate the cost of reusing Ada generics as characterized in section 5.1.

The general evaluation goals are (i) characterize the degree of discrepancies between a given reuse specification (see Table 3) and a given reuse candidate (Table 2), and (ii) what is the cost of bridging the gap between before–reuse and after–reuse characteristics. The first type of evaluation goal can be achieved by capturing detailed information with respect to the object–before–reuse and object–after–reuse dimensions. The second goal requires the inclusion of data characterizing the reuse process itself and past experience about similar reuse activities.

We use the goal/question/metric paradigm to perform the above kind of goal–oriented evaluation [6, 8, 10]. It provides templates for guiding the selection of appropriate metrics based on a precise definition of the evaluation goal. Guidance exists at the level of identifying certain types of metrics (e.g., to quantify the object of interest, to quantify the perspective of interest, to quantify the quality aspect of interest). Using the goal/question/metric paradigm in conjunction with reuse characterizations like the ones depicted in Tables 2, 3, and 4, provides very detailed guidance as to what exact metrics need to be used. For example, evaluation of the Ada generic example suggests metrics to characterize discrepancies between the desired object–after–reuse and all before–reuse candidates in terms of (i) function, use, type, granularity, and representation on a nominal scale defined by the respective categories, (ii) input/output interface on an ordinal scale

'number of instantiation params', (iii) application and solution domains on nominal scales, and (iv) qualities such as performance based on benchmark tests.

5.4. Planning the Population of Reuse Repositories

We will demonstrate the benefits of our reuse characterization scheme to populate a reuse repository with generic Ada packages as characterized in section 5.1.

Reuse is economical from a project perspective if the effort required to bridge the gap between an object–before–reuse (available in some experience base) and the desired object–after–reuse is less than the effort required to create the object–after–reuse from scratch. Reuse is economical from an organization's perspective if the effort required for creating the reuse repository is less than the sum of all project–specific savings based on reuse.

Based on the above statement, populating a reuse repository constitutes an optimization problem for the organization. For example, high effort for populating a reuse repository may be justified if (i) small savings in many projects are expected, or (ii) large savings in a small number of projects are expected. For example, object 'buffer.ada' could have been transformed to adhere to object oriented principles prior to introducing it into the repository. This would have excluded the project specific risk and cost.

The cost of reusing an object–before–reuse from an experience base depends on its distance to the desired object–after–reuse and the mechanisms employed to bridge that distance. The cost of populating a reuse repository depends on how much effort is required to transform existing objects into objects–before–reuse. Both efforts together are aimed at bridging the gap between the project in which some objects were produced and the projects in which they are intended to be reused. The inclusion of a generic package 'buffer.ada' into the repository instead of specific instances 'integer_buffer.ada' and 'real–buffer.ada' requires some up–front transformation (i.e., abstraction). The advantage of creating an object 'buffer.ada' is that it reduces the project–specific cost of creating object 'string_buffer.ada' (or any other buffer for that matter) and

quantifies the cost of modification.

Finding the appropriate characteristics for objects–before–reuse to minimize project–specific reuse costs requires a good understanding of future reuse needs (objects–after–reuse) and the reuse processes to be employed (reuse process). The more one knows about future reuse needs within an organization, the better job one can do of populating a repository. For example, the object–before–reuse characteristics of Ada generics in Table 2 were derived from the corresponding object–after–reuse and reuse process characteristics in Tables 3 and 4. It would have made no sense to include Ada generics into the experience base that (i) are not based on the same instantiation parameters as all anticipated objects–after–reuse because modification is assumed via parameterized instantiation, (ii) do not exhibit high reliability and performance, and (iii) have not the same solution domain except we understand the implication of different solution domains. Without any knowledge of the object–after–reuse and reuse process characteristics, the task of populating a reuse repository is about as meaningful as investing in the mass–production of concrete components in the area of civil engineering without knowing whether we want to build bridges, town houses or high–rise buildings.

6. A REUSE–ORIENTED SOFTWARE ENVIRONMENT MODEL

Effective reuse according to the reuse–oriented software development model depicted in Figure 2 of Section 2 needs to take place in an environment that supports continuous improvement, i.e., recording of experience across all projects, appropriate packaging and storing of recorded experience, and reusing existing experience whenever feasible. Figure 6 depicts such an environment model.

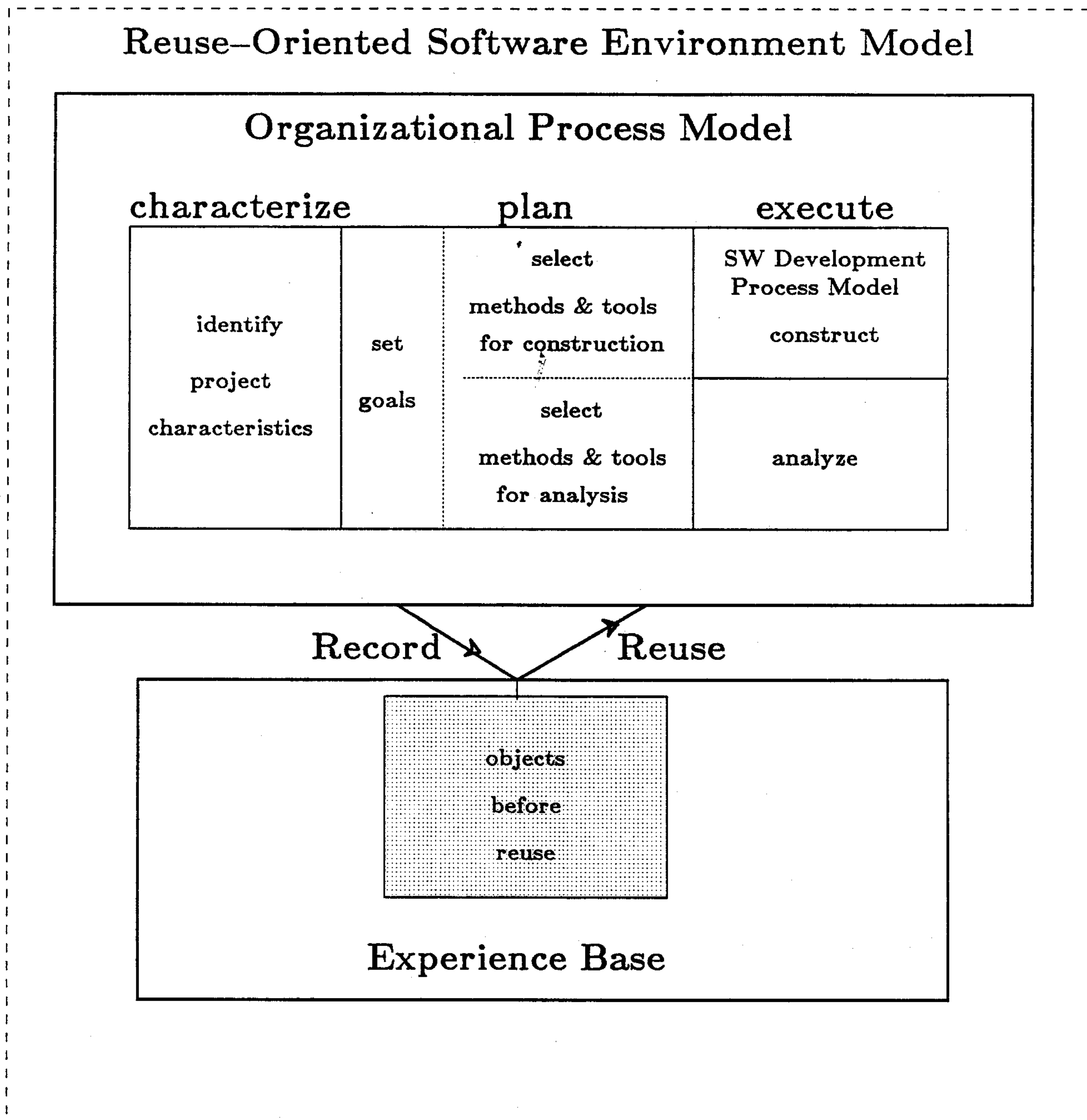

Figure 6: Reuse–Oriented Software Environment Model

Each project is performed according to an organization process model based on the improvement paradigm [2, 5]:

1. **Characterize**: Identify characteristics of the current project environment so that the

appropriate past experience can be made available to the current project.

2. **Plan:** (A) Set up the goals for the project and refine them into quantifiable questions and metrics for successful project performance and improvement over previous project performances (e.g., based upon the goal/question/metric paradigm [6]).

   (B) Choose the appropriate software development process model for this project with the supporting methods and tools – both for construction and analysis.

3. **Execute:** (A) Construct the products according to the chosen development process model, methods and tools.

   (B) Collect the prescribed data, validate and analyze it to provide feedback in real–time for corrective action on the current project.

4. **Feedback:** (A) Analyze the data to evaluate the current practices, determine problems, record findings and make recommendations for improvement for future projects.

   (B) Package the experiences in the form of updated and refined models and other forms of structured knowledge gained from this and previous projects, and save it in an experience base so it can be available to future projects.

The experience base is not a passive entity that simply stores experience. It is an active organizational entity in the context of the reuse–oriented environment model which – in addition to storing experience in a variety of repositories – involves the constant modification of experience to increase its reuse potential. It plays the role of an organizational "server" aimed at satisfying project–specific requests effectively. The constant collection of measurement data regarding objects–after–reuse and the reuse processes themselves enables the judgements needed to populate the experience base effectively and to select the best suited objects–before–reuse to satisfy project–specific reuse needs based upon experiences. The organizational process model based on the improvement paradigm supports the integration of measurement–based analysis and construction.

For more detail about the reuse–oriented environment model, the reader is referred to [7].

7. CONCLUSIONS

The model–based reuse characterization scheme introduced in this paper has advantages over existing schemes in that it (a) allows us to capture the reuse of any type of experience, (b) distinguishes between objects–before–reuse, objects–after–reuse, and the reuse process itself, and (c) provides a rationale for the chosen characterizing dimensions. In the past most the scope of reuse schemes was limited to objects–before–reuse.

We have demonstrated the advantages of such a model–based scheme by applying it to the characterization of example reuse scenarios. Especially its usefulness for evaluating the cost of reuse and planning the population of reuse repositories were stressed.

Finally, we gave a model how we believe reuse should be integrated into an environment aimed at continuous improvement based on learning and reuse. A specific instantiation of such an environment, the 'code factory', is currently being developed at the University of Maryland [12]. In order to make reuse a reality, more research is required towards understanding and conceptualizing activities and aspects related to reuse, learning and the experience base.

8. ACKNOWLEDGEMENTS

We thank all our colleagues and graduate students who contributed to this paper, especially all members of the TAME and CARE project.

9. REFERENCES

[1] V. R. Basili, "Can We Measure Software Technology: Lessons Learned from Eight Years of Trying", in Proc. Tenth Annual Software Engineering Workshop, NASA Goddard Space Flight Center, Greenbelt, MD, December 1985.

[2] V. R. Basili, "Quantitative Evaluation of Software Methodology", Dept. of Computer Science, University of Maryland, College Park, TR-1519, July 1985 [also in Proc. of the First Pan Pacific Computer Conference, Australia, September 1986].

[3] V. R. Basili, "Viewing Maintenance as Reuse-Oriented Software Development", IEEE Software Magazine, January 1990, pp. 19–25.

[4] V. R. Basili and H. D. Rombach, "Tailoring the Software Process to Project Goals and Environments", Proc. of the Ninth International Conference on Software Engineering, Monterey, CA, March 30 - April 2, 1987, pp. 345–357.

[5] V. R. Basili and H. D. Rombach, "TAME: Integrating Measurement into Software Environments", Technical Report TR-1764 (or TAME-TR-1-1987), Dept. of Computer Science, University of Maryland, College Park, MD 20742, June 1987.

[6] V. R. Basili and H. D. Rombach "The TAME Project: Towards Improvement-Oriented Software Environments", IEEE Transactions on Software Engineering, vol. SE-14, no. 6, June 1988, pp. 758–773.

[7] V. R. Basili and H. D. Rombach, "Towards a Comprehensive Framework for Reuse: A Reuse-Enabling Software Evolution Environment", Technical Report (UMIACS-TR-88-92, CS-TR-2158), Department of Computer Science, University of Maryland, College Park, MD 20742, December 1988.

[8] V. R. Basili and R. W. Selby, "Comparing the Effectiveness of Software Testing Strategies", IEEE Transactions on Software Engineering, vol.SE-13, no.12, December 1987, pp.1278–1296.

[9] V. R. Basili and M. Shaw, "Scope of Software Reuse", White paper, working group on 'Scope of Software Reuse', Tenth Minnowbrook Workshop on Software Reuse, Blue Mountain Lake, New York, July 1987 (in preparation).

[10] V. R. Basili and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data", IEEE Transactions on Software Engineering, vol.SE-10, no.3, November 1984, pp.728–738.

[11] Ted Biggerstaff, "Reusability Framework, Assessment, and Directions", IEEE Software Magazine, March 1987, pp.41–49.

[12] G. Caldiera and V. R. Basili, "Reengineering Existing Software for Reusability", Technical Report (UMIACS-TR-90-30, CS-TR-2419), Department of Computer Science, University of Maryland, College Park, MD 20742, February 1990.

[13] P. Freeman, "Reusable Software Engineering: Concepts and Research Directions", Proc. of the Workshop on Reusability, September 1983, pp. 63–76.

[14] R. Prieto-Diaz and P. Freeman, "Classifying Software for Reusability", IEEE Software, vol.4, no.1, January 1987, pp. 6–16.

[15] IEEE Software, special issue on 'Reusing Software', vol.4, no.1, January 1987.

[16] IEEE Software, special issue on 'Tools: Making Reuse a Reality', vol.4, no.7, July 1987.

[17] G. A. Jones and R. Prieto-Diaz, "Building and Managing Software Libraries", Proc. Compsac'88, Chicago, October 5–7, 1988, pp. 228–236.

[18] A. Kouchakdjian, V. R. Basili, and S. Green, "The Evolution of the Cleanroom Process in the Software Engineering Laboratory", IEEE Software Magazine (to appear 1990).

[19] F. E. McGarry, "Recent SEL Studies", in Proc. Tenth Annual Software Engineering Workshop, NASA Goddard Space Flight Center, Greenbelt, MD, Dec. 1985.

[20] R. W. Selby, Jr., V. R. Basili, and T. Baker, "CLEANROOM Software Development: An Empirical Evaluation", IEEE Transactions on Software Engineering, vol. SE–13, no. 9, September 1987, pp.1027–1037.

[21] Mary Shaw, "Purposes and Varieties of Software Reuse", Proceedings of the Tenth Minnowbrook Workshop on Software Reuse, Blue Mountain Lake, New York, July, 1987.

[22] T. A. Standish, "An Essay on Software Reuse", IEEE Transactions on Software Engineering, vol. SE–10, no. 5, September 1984, pp.494–497.

[23] W. Tracz, "Tutorial on 'Software Reuse: Emerging Technology'", IEEE Catalog Number EHO278–2, 1988.

[24] J. Valett, B. Decker, J. Buell, "The Software Management Environment", in Proc. Thirteenth Annual Software Engineering Workshop, NASA/Goddard Space Flight Center, Greenbelt, MD, November 30, 1988.

[25] M. V. Zelkowitz (ed.), "Proceedings of the University of Maryland Workshop on 'Requirements for a Software Engineering Environment', Greenbelt, MD, May 1986", Technical Report TR–1733, Dept. of Computer Science, University of Maryland, College Park, MD 20742, December 1986 [to be published as a book, Ablex Publ., 1988].