Support for comprehensive reuse

by V.R. Basili and H.D. Rombach

Reuse of products, processes and other knowledge will be the key to enable the software industry to achieve the dramatic improvement in productivity and quality required to satisfy anticipated growing demands. Although experience shows that certain kinds of reuse can be successful, general success has been elusive. A software life-cycle technology that allows comprehensive reuse of all kinds of software-related experience could provide the means of achieving the desired order-of-magnitude improvements. In this paper, we introduce a comprehensive framework of models, model-based characterisation schemes, and support mechanisms for better understanding, evaluating, planning and supporting all aspects of reuse.

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 be improved by reusing all forms of proven experience, including products and processes, as well as quality and productivity models. Productivity should increase by using existing experience, rather than creating everything from the beginning.

Reusing existing experience is a key ingredient to progress in any discipline. Without reuse everything must be relearnt and recreated; progress in an economical fashion is unlikely. Reuse is less institutionalised in software engineering than in any other engineering discipline. Nevertheless, there are successful cases of reuse, i.e. product reuse. The potential pay-off from reuse can be quite high in software engineering, as it is inexpensive to store and reproduce software engineering experience compared to other disciplines.

The goal of research in the area of reuse is to develop and support systematic approaches for effectively reusing existing experience, in order to maximise quality and productivity. A number of different reuse approaches have appeared in the literature [1–10].

This paper presents a comprehensive framework for reuse, consisting of a reuse model, characterisation schemes based on this model, the improvement-oriented TAME environment model describing the integration of reuse into the (reuse) enabling software development processes, mechanisms needed to support comprehensive reuse in the context of the TAME environment model, and (partial) prototype implementations of the TAME environment model. From a number of important assumptions regarding the nature of software development and reuse, we derive four essential requirements for any useful reuse model and related characterisation scheme. We illustrate that existing models and characterisation schemes only partially satisfy these essential requirements. We introduce a new reuse model, which is comprehensive in the sense that it satisfies all four reuse requirements, and use it to derive a reuse characterisation scheme. Finally, we point out the mechanisms needed to support effective reuse according to this model. Throughout the paper, we use examples of reusing *generic Ada packages*, *design inspections* and *cost models* to illustrate our approach.

2 Scope of comprehensive reuse

The reuse framework 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 fifteen years' work of analysing software processes and products [11–16]. From these assumptions, we derive four essential requirements for any useful reuse model and related characterisation scheme.

2.1 Software development assumptions

According to a common software development project model depicted in Fig. 1, the goal of software development is to produce project deliverables (i.e. project output) that satisfy project requirements (i.e. project input). This goal is achieved according to a development process model that coordinates the interaction between available personnel, practices, methods and tools [17].

With regard to software development we make the following assumptions.

- **Software development needs to be viewed as an 'experimental' discipline.** An evolutionary model is needed that enables organisations to learn from each development and incrementally improve their ability to engineer quality software products. Such a model requires the ability to define project goals; select and tailor the appropriate process models, practices, methods and techniques; and capture the experiences gained from each

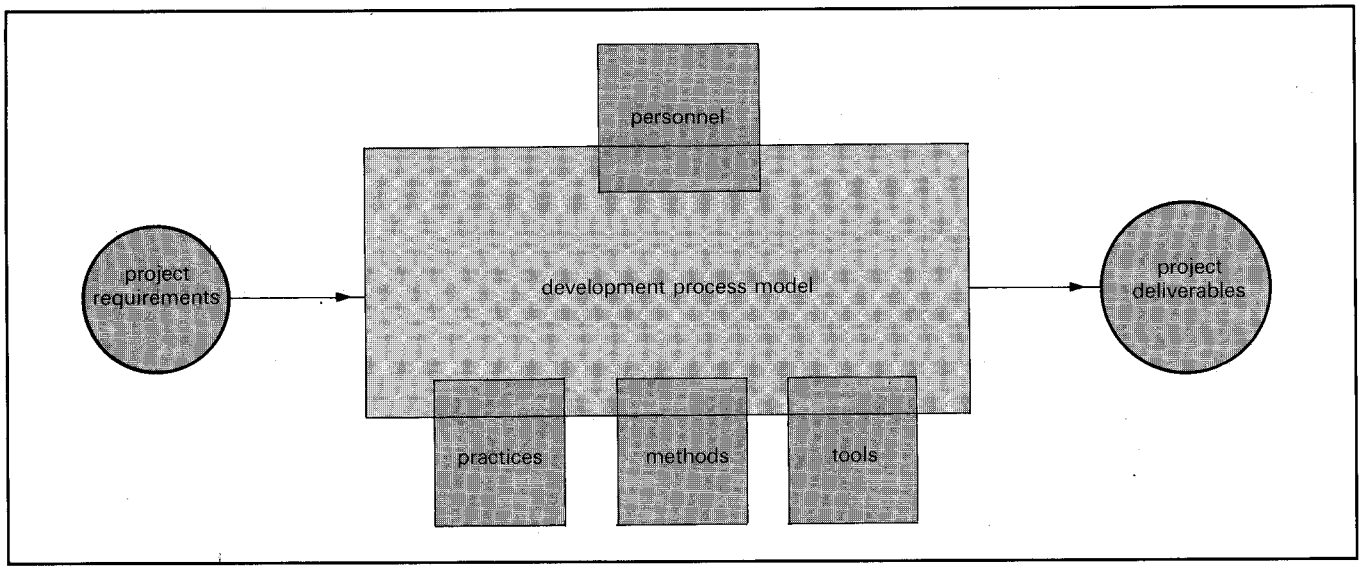

Fig. 1 Software development project model

project in reusable form. Measurement is essential.

- **A single software development approach cannot be assumed for all software development projects.** Different project requirements and other project characteristics may suggest and justify different approaches. 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.
- **Existing software development approaches need to be tailorable to project requirements and characteristics.** In order to reuse existing development process models, practices, methods and tools across projects with different requirements and characteristics, they need to be tailorable.

2.2 Software reuse assumptions

Reuse-oriented software development assumes that, given the project-specific requirements $\bar{x}$ for an object x, we consider reusing an already existing object x_k instead of creating x from the beginning. Reuse involves identifying a set of reuse candidates $x_1, \ldots, x_n$ from an experience base, evaluating their potential for satisfying $\bar{x}$, selecting the best-suited candidate x_k and, if required, modifying the selected candidate x_k into x. Related issues have been discussed in Reference 18. In the case of reuse-oriented development, $\bar{x}$ is not only the specification for the needed object x, but also the specification for all the above-mentioned reuse activities.

As we learn from each project which kinds of experience are reusable and why, we can establish better criteria for what should and what should not be made available in the experience base. The term 'experience base' suggests that we expect storage of all kinds of software-related experience, not just products. The experience base can be improved from inside, as well as outside. From inside, we can record experience from ongoing projects, which satisfies current reuse criteria for future reuse, and we can repackage existing experience, through various mechanisms, in order to better satisfy our current reuse criteria. From outside, we can infuse experience that exists outside the organisation in the experience base. It is important to note that the remainder of this paper deals only with the reuse of experience available in an experience base and the improvement of such an experience base from inside (shaded section in Fig. 2).

With regard to software reuse we make the following assumptions.

☐ **All experience can be reused.** Traditionally, the emphasis has been on reusing concrete objects of type 'source code'. This limitation reflects the traditional view that software equals code. It ignores the importance of reusing all kinds of software-related experience, including products, processes and other knowledge. The term 'product' refers to either a concrete document or artifact created during a software project, or a product model describing a class of concrete documents or artifacts with common characteristics. The term 'process' refers to either a concrete activity of action (performed by a human being or a machine) aimed at creating some software product, or a process model describing a class of activities or actions with common characteristics. The phase 'other knowledge' refers to anything useful for software development, including quality and productivity models or models of the application being implemented.

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 mechanism. The reuse of 'design inspections' represents an example of process reuse. Design inspections are off-line fault detection and isolation methods applied during the component design phase. They can be based on different techniques for reading (e.g. ad hoc, sequential, control-flow-oriented, step-wise 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).

☐ **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 previous projects must be antici-

pated. The degree of modification depends on how many, and to what degree, existing object characteristics differ from those required. The time of modification depends on when the reuse requirements for a project or class of projects are known. Modification can take place as part of actual reuse (i.e. as part of the modification activity of the reuse process model in Fig. 2) and/or before actual reuse (i.e. as part of the repackaging activity in Fig. 2).

To reuse an Ada package 'list of integers' in order to organise 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 characterised 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 altogether (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 characterised 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 redefining 'estimated product size' to be measured in terms of 'function points' instead of 'lines of code', or recomputing 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.

□ **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 pay-off. Reuse pay-off is not always easy to evaluate [19]. We need to understand the reuse requirements; how well the available reuse candidates are qualified to meet these requirements; and the mechanisms available to perform the necessary modification.

Assume the existence of a set of Ada generics that 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 previous 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

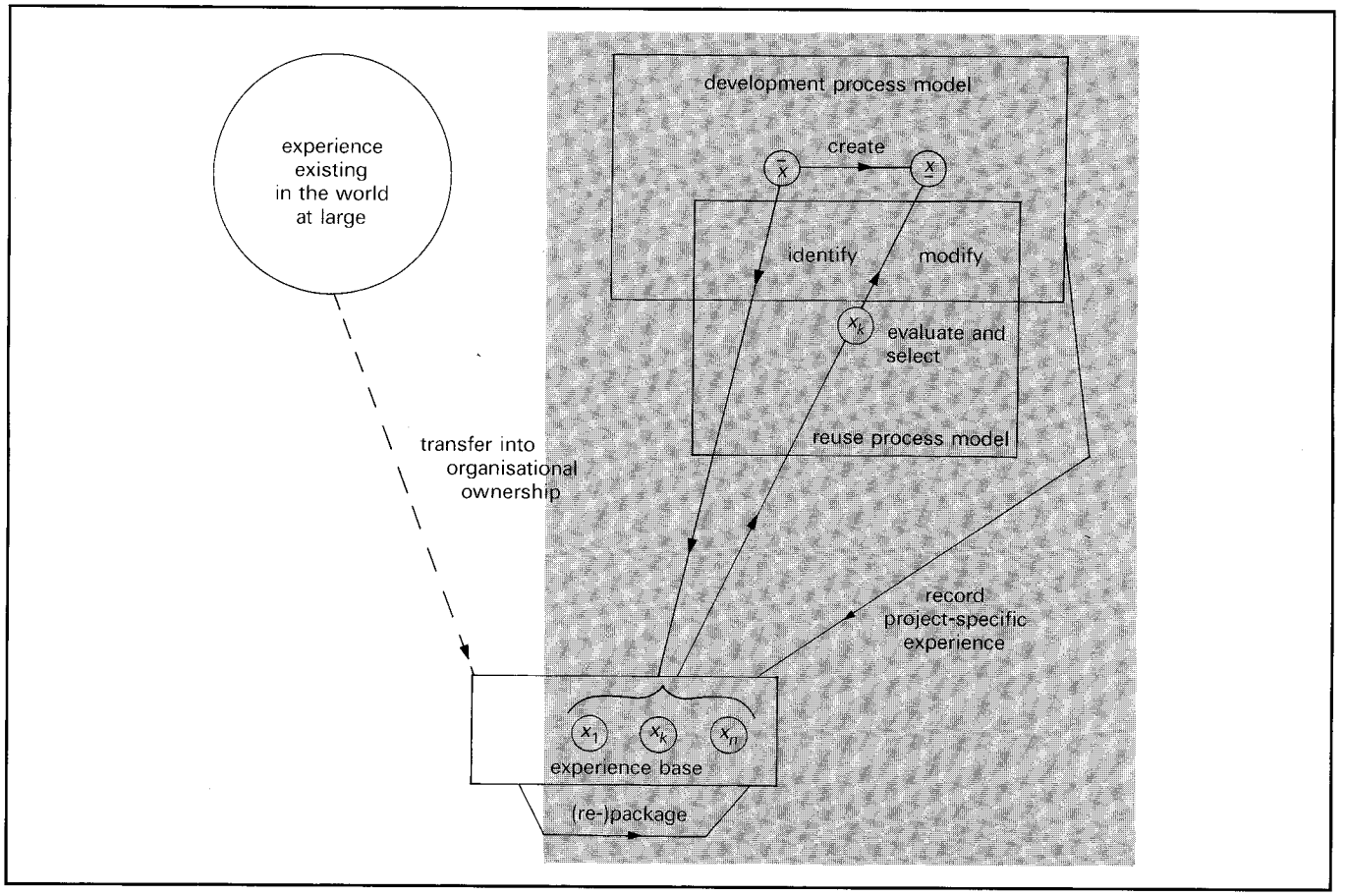

Fig. 2 Reuse-oriented software development model

method [20, 21]. *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 step-wise 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 step-wise abstraction-oriented reading technique* [22].

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 developments. 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 recalibrated. This question can only be answered by thorough analysis of a number of Cleanroom projects.

□ **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 to 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 that 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 during 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 components at the end of component design, once for all components of a subsystem, or once for each component). 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 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 model requirements*

The above software reuse assumptions suggest that reuse is a complex concept. We need to build models and characterisation schemes that allow us to define and understand, compare and evaluate, and plan the reuse requirements, the reuse candidates, the reuse process itself and the potential for effective reuse. Based on the above assumptions, such models and characterisation schemes need to satisfy the following four requirements:

- **applicable to all types of reuse objects;** we want to be able to include products, processes and all other kinds of knowledge, such as quality and productivity models.
- **capable of modelling reuse candidates and reuse requirements;** we want to be able to capture the reuse candidates, as well as the reuse requirements in the current project. This will enable us to judge the suitability of a given reuse candidate, based on the distance between the characteristics of the reuse requirements and the reuse candidate, and establish criteria for useful reuse candidates based on anticipated reuse requirements.
- **capable of modelling the reuse process itself;** we want to be able to judge the ease of bridging the gap between different characteristics of reuse candidates and reuse requirements, and derive additional criteria for useful reuse candidates, based on characteristics of the reuse process itself.
- **defined and rationalised, so that they can be easily tailored to specific project requirements and characteristics;** we want to be able to adjust a given reuse model and characterisation scheme to changing project requirements and characteristics in a systematic way. This not only requires the ability to change the scheme, but also some kind of rationale that ties the given reuse characterisation scheme 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 Existing reuse models

A number of research groups have developed (implicit) models and characterisation schemes for reuse [2–4, 8, 9]. The schemes can be distinguished as *special-purpose schemes* and *meta schemes*.

The large majority of published characterisation schemes have been developed for a special purpose. They consist of a fixed number of characterisation dimensions. Their intention is to characterise software products as they exist. Typical dimensions for characterising source code objects in a repository are 'function', 'size' or 'type of problem'. Example schemes include the schemes published in References 3 and 4, the ACM Computing Reviews Scheme, the AFIPS Taxonomy of Computer Science and Engineering, schemes for functional collections (e.g. GAMS, SHARE, SSP, SPSS, IMSL) and schemes for commercial software cata-

* Definitions of semantic closeness can be derived from existing work [23].

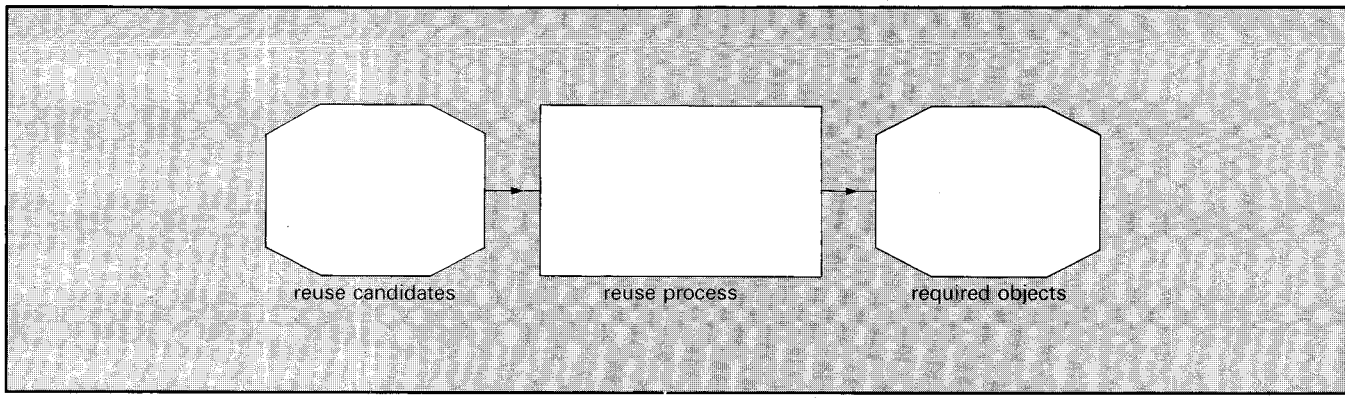

Fig. 3 Abstract reuse model (refinement level 0)

logues (e.g. ICP, IDS, IBM Software Catalog, Apple Book). It is obvious that special-purpose schemes are not designed to satisfy the reuse modelling requirements of Section 2.3.

A few characterisation schemes can be instantiated for different purposes. They explicitly acknowledge the need for different schemes (or the expansion of existing ones) due to the different or changing requirements of an organisation. They therefore allow the instantiation of any imaginable scheme. An excellent example is Prieto-Diaz's facet-based meta-characterisation scheme [5, 24]. 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 has not been utilised. Instead, most examples focus on product reuse, are limited to the reuse candidates and ignore the reuse process entirely. Meta schemes were not designed to satisfy the reuse modelling requirements of Section 2.3.

To illustrate the capabilities of existing schemes, we give the following instance of an example meta scheme:†

☐ **name:** what is the product's name? (e.g. buffer.ada, queue.ada, list.pascal)

☐ **function:** what is the functional specification or purpose of the product? (e.g. integer_queue, ⟨element⟩_buffer, sensor control system)

☐ **type:** what type of product is it? (e.g. requirements document, design document, code document)

☐ **granularity:** what is the product's scope? (e.g. system level, subsystem level, component-package, procedure, function-level)

☐ **representation:** how is the product represented? (e.g. informal set of guidelines, schematised templates, languages such as Ada)

☐ **input/output:** what are the external input/output dependencies of the product 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)

☐ **application domain:** what application classes was the product developed for? (e.g. ground support software for satellites, business software for banking, payroll software)

This scheme is applicable to all reuse product candidates. For example, a generic Ada package 'buffer.ada' may be characterised as having identifier 'buffer.ada', offering the function '⟨element⟩_buffer', being usable as a 'product' of type 'code document' at the 'package component level', and being represented in 'Ada'. A self-contained definition of a package requires knowledge regarding the instantiation parameters, as well as its visibility of externally 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 Ada language and may 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'. In order to characterise reuse candidates of type process or knowledge, new categories need to be generated.

Such schemes have typically been used to characterise reuse candidates only. However, in order to evaluate the reuse potential of a reuse candidate in a given reuse scenario, we need to understand the distance between its characteristics and the stated or anticipated reuse requirements. 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 needs and reuse candiates, it is hard to predict the cost involved in reusing a particular object and to establish criteria for populating a reuse repository that supports cost-effective reuse.

This scheme provides no information for characterising the reuse process. To accurately predict the cost of reuse, we not only have to understand the distance to be bridged between reuse candidates and reuse requirements, but also the intended process to bridge it (i.e. the reuse process). For example, we may expect that it is easier to bridge the distance with respect to function by using a parameterised instantiation mechanism, rather than modifying the existing package by hand.

There is no explicit rationale for the eight 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 as to tailoring the example scheme to new requirements with respect to what is to be changed (e.g. only some categories, dimensions or the entire implicitly underlying model), or how it is to be changed. For example, it is not clear what needs to be changed in order to make the above scheme applicable to reuse candidates of type process or knowledge.

† Characterisation dimensions are marked with ☐; example categories for each dimension are listed in parentheses.

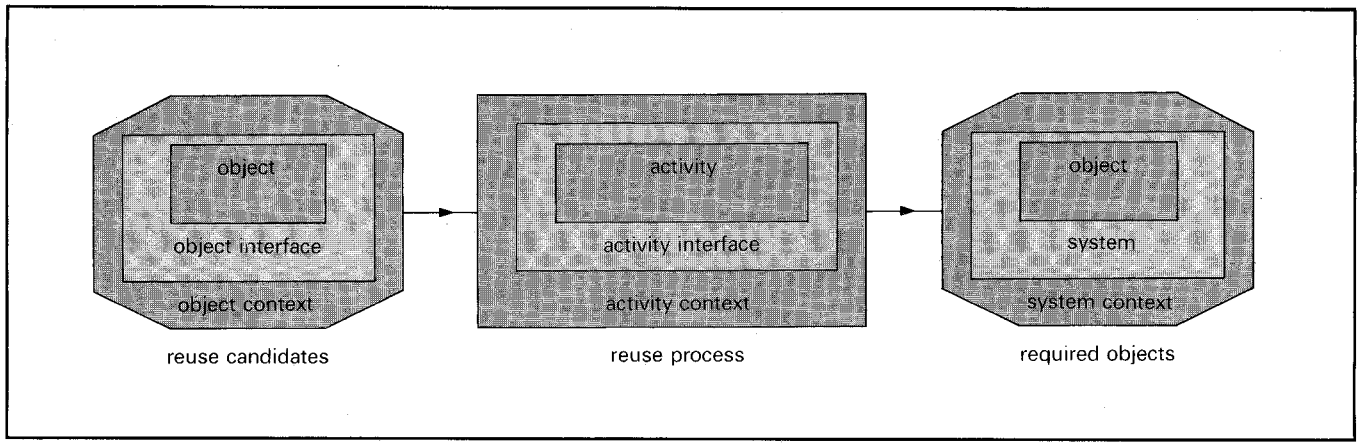

Fig. 4 Reuse model (refinement level 1)

In summary, existing schemes (special-purpose as well as meta schemes) only partially satisfy the requirements laid out in Section 2.3. The most crucial shortcoming is the lack of rationales, which makes it hard to tailor schemes to changing requirements and environment characteristics. This observation suggests the need for new, broader reuse models and characterisation schemes. In the next Section, we suggest a comprehensive reuse model and characterisation schemes that satisfy all four reuse requirements.

4 A comprehensive reuse model

In this Section, we define a comprehensive reuse model and characterisation schemes that satisfy the reuse requirements stated in Section 2.3. We start with a very general reuse model, refine it, step by step, until it generates reuse characterisation dimensions at the level of detail needed to understand, evaluate, motivate and improve reuse. The stepwise refinements of the high-level reuse model capture the rationale of the resulting reuse characterisation scheme. Such a rationale is important for understanding, motivating and tailoring any reuse characterisation scheme.

4.1 Reuse model

The comprehensive reuse model used in this Section is consistent with the view of reuse represented in Section 2.2. Reuse comprises the transformation of existing reuse candidates into required objects which satisfy established reuse needs (Fig. 3). The transformation is referred to as reuse process. Specifications of the required objects are an essential part of the reuse requirements that guide any reuse process.

The reuse candidates represent experience from the same project, previous projects or other sources, which have been evaluated as being of potential reuse value, and have been made available in some form of experience base. The reuse requirements specify objects required in the current project. In the case of successful reuse, these required objects would be the potentially modified versions of existing reuse candidates. Both the reuse candidates and reuse requirements may refer to any type of experience accumulated in the context of software projects, ranging from products to processes to knowledge. The reuse process transforms reuse candidates into objects that satisfy given reuse requirements.

In order to better understand reuse-related issues, we refine each component of the reuse model further. The result of this first refinement step is depicted in Fig. 4.

Each *reuse candidate* is a specific *object* considered for reuse. The object has various attributes that describe and bound it. 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 an environment that leaves its characteristics on the object, although those characteristics may not be visible. We call this the *object context*.

Given *reuse requirements* may be satisfied by a set of reuse candidates. Therefore, we may have to consider different attributes for each required object. 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 a reuse candidate from a repository, based on the characteristics of the known reuse needs, 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 organisational support provided for this experience transfer is referred to as *activity context*.

Based on the goals for the specific project, as well as the organisation, we must assess the required qualities of the reused object as stated by the reuse requirements; the quality of the reuse process, especially its integration into the enabling software evolution process; and the quality of the existing reuse candidates.

4.2 Model-based reuse characterisation scheme

Each component of the first model refinement (Fig. 4) is further refined, as depicted in Figs. 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.

4.2.1 Reuse candidates: in order to characterise the object itself, we have chosen to provide the following 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. component) and its representaton (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 dimensions provide us with historical information, such as the application classes for which the object was developed (e.g. ground support software for satellites), the environment in which the object was developed (e.g. waterfall life-cycle model) and its validated or anticipated quality (e.g. reliability). The resulting model refinement is depicted in Fig. 5*a*.

Each reuse candidate is characterised 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, graphics tool, process model, cost model)
- **granularity:** what is the object's scope? (e.g. system level, subsystem level, component-package, procedure, function-level, entire life-cycle, design stage, coding stage)
- **representation:** how is the object represented? (e.g. data, informal set of guidelines, schematised templates, formal mathematical model, languages such as Ada, automated tools)
- **input/output:** what external input/output dependencies of the object are required 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 additional assumptions and dependencies are needed to understand the object? (e.g. assumption about 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:** in what environment classes was the object developed? (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, predicability)

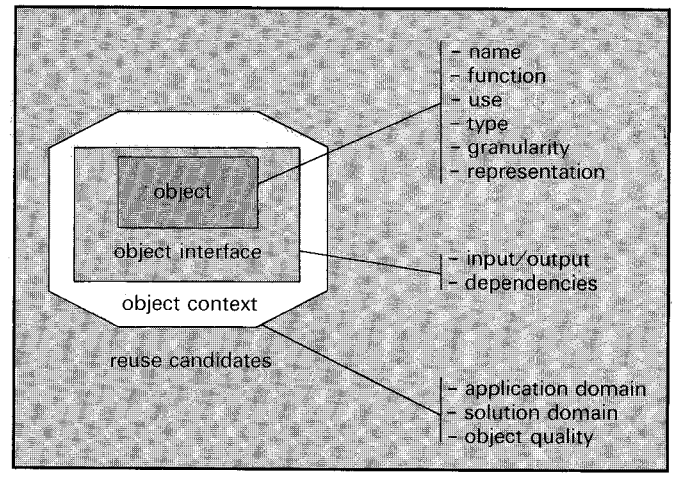

Fig. 5*a* Reuse model (reuse candidates/refinement level 2)

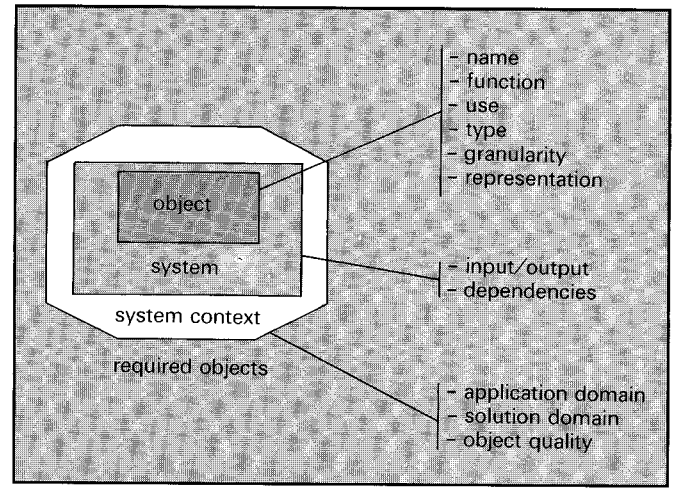

Fig. 5*b* Reuse model (reuse requirements/refinement level 2)

A subset of this scheme has been used in Section 3. In contrast, we now have a rationale for these dimensions (Fig. 5*a*)

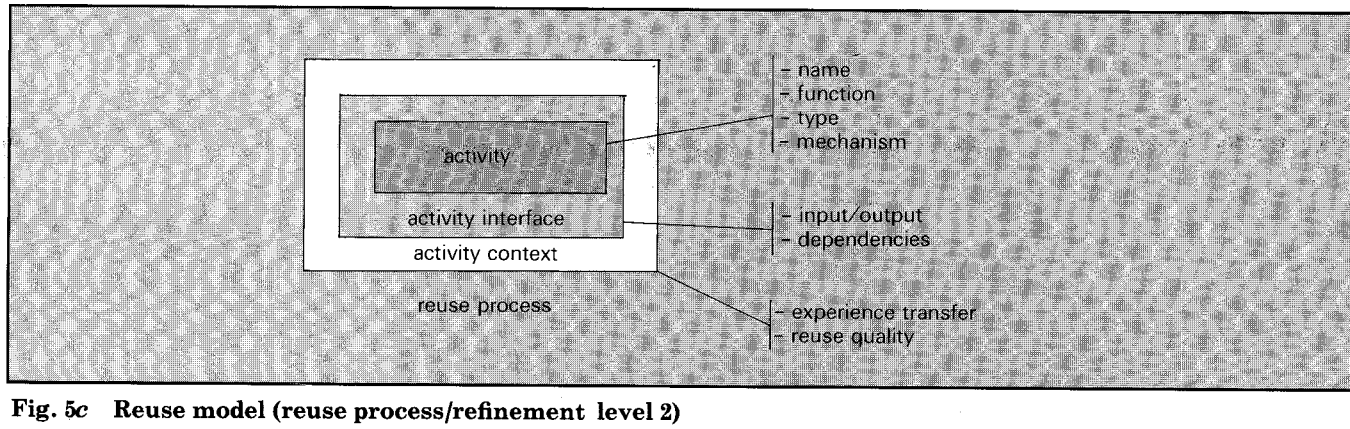

Fig. 5*c* Reuse model (reuse process/refinement level 2)

and understand that they cover only part (i.e. the reuse candidate) of the comprehensive reuse model depicted in Fig. 4.

4.2.2 Required objects: in order to characterise the required objects (or reuse requirements), we have chosen the same dimensions and supporting categories as for the reuse candidates. The resulting model refinement is depicted in Fig. 5*b*.

However, an object may change its characteristics during the actual process of reuse. Therefore, its characterisations before and after reuse can be expected to be different. For example, a reuse candidate may be a compiler (type) product (use) and may have been developed according to a waterfall life-cycle approach (solution domain). The required object is 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 reuse candidates and requirements objects, there is 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 a reuse candidate and the required object give an indication of the gap to be bridged in the event of reuse.

4.2.3 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 characterise each reuse activity, we may be interested in its name (e.g. modify.pl), its function (e.g. modify an identified reuse candidate to entirely satisfy given reuse requirements), its type (e.g. identification, evaluation, modification) and the mechanism used to perform its function (e.g. modification via parameterisation). The interface of each activity may consist of such things as the explicit input/output interfaces between the activity and the enabling software evolution environment (e.g. in the case of modification performed during the coding phase, assumes the eixstence of a specification), and other assumptions regarding the evolution environment that need to be satisfied (e.g. existence of certain configuration control policies). The activity context may include information about how reuse candidates are transferred to satisfy given reuse requirements (experience transfer) and the quality of each reuse activity (e.g. reliability, productivity). This refinement of the reuse process is depicted in Fig. 5*c*.

In more detail, the dimensions and example categories for each reuse activity are

- **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 $\{x_i\}$ that satisfy certain characteristics of the reuse requirements $\bar{x}$; evaluate the potential of the selected candidate objects of satisfying the given system and system context dimensions of the reuse requirements $\bar{x}$ 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 judgment, by evaluation of historical base-line measurement data; in the case of modification, e.g. verbatim, parameterised, template-based, unconstrained; in the case of integration, e.g. according to the system configuration plan, according to the project/process plan)
- **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. description of reuse requirements/set of reuse candidates; in the case of modification, e.g. specification of the object to be reused/object to be reused)
- **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)
- **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)

4.3 Example applications of the comprehensive reuse model

We demonstrate the applicability of our model-based reuse scheme by characterising the three hypothetical reuse scenarios that have been used informally throughout this paper: Ada generics, design inspections and cost models. The resulting characterisations are summarised in Tables 1–3.

5 Support mechanisms for comprehensive reuse

According to the reuse-oriented software development model depicted in Fig. 2, effective reuse 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. In the TAME project at the University of Maryland, such an environment model has been proposed, and (partial) prototype environments are currently being built according to this model [15]. In the remainder of this Section, we introduce the reuse-oriented TAME environment model, discuss a number of mechanisms for effective reuse and introduce several prototype environments being built according to the TAME model.

5.1 The reuse-oriented TAME environment model

The important components of the reuse-oriented TAME

environment model are depicted in Fig. 6: the project organisation, which performs individual development projects; and the experience factory, which stores and actively modifies development experience from all projects. The shaded areas in Fig. 6 indicate how the reuse model of Fig. 3 intersects with the TAME environment model.

Within the project organisation, each development project is performed according to the quality improvement paradigm [15, 25]. The quality improvement paradigm consists of the following steps:

- **plan:** characterise the current project environment so that the appropriate past experience can be made available to the current project. 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 on the goal/question/metric paradigm [15, 26]). Choose the appropriate software development process model for this project with the supporting methods and tools, for both construction and analysis.
- **execute:** construct the products according to the chosen development process model, methods and tools. Collect the prescribed data, validate and analyse it to provide feedback in real-time for corrective action on the current project.
- **package:** analyse the data in a post-mortem fashion to evaluate the current practices, determine problems, record findings and make recommendations for improvement for future projects. 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 that it can be available to future projects.

The experience base contains reuse candidates of different types, granularity and representation. Example entries in the case of the examples described in Section 4.3 include objects of type 'code document', granularity 'package' and representation 'Ada'; objects of type 'inspection method', granularity 'design stage' and representation 'schematised template'; and objects of type 'cost model', granularity 'entire life-cycle' and representation 'formal mathematical model'.

During each step of a development project performed according to the quality improvement paradigm, reuse requirements are identified and matches made against reuse candidates available in the experience base. During the characterisation step, characteristics of the current project environment can be used to identify appropriate past experience in the experience base, e.g. based on project characteristics, the appropriate instantiation of a cost model can be generated. During the planning step, project goals can be used to identify existing similar goal/question/metric models or process/product/quality models in the experience base, e.g. based on project goals, a goal/question/metric model can be chosen for evaluating a design inspection method. During the execution step, product specifications can be used to identify existing components from previous projects, such as Ada generics. During the feedback step, the analysis goals generated during planning are used as the basis of analysis by fitting base-lines to compare against the current data. As part of the feedback step a decision is made as to which experiences are worth recording. The degree of guidance that can be provided for entering reuse candidates into the experience base depends on the accumulated knowledge of expected reuse requests for future projects.

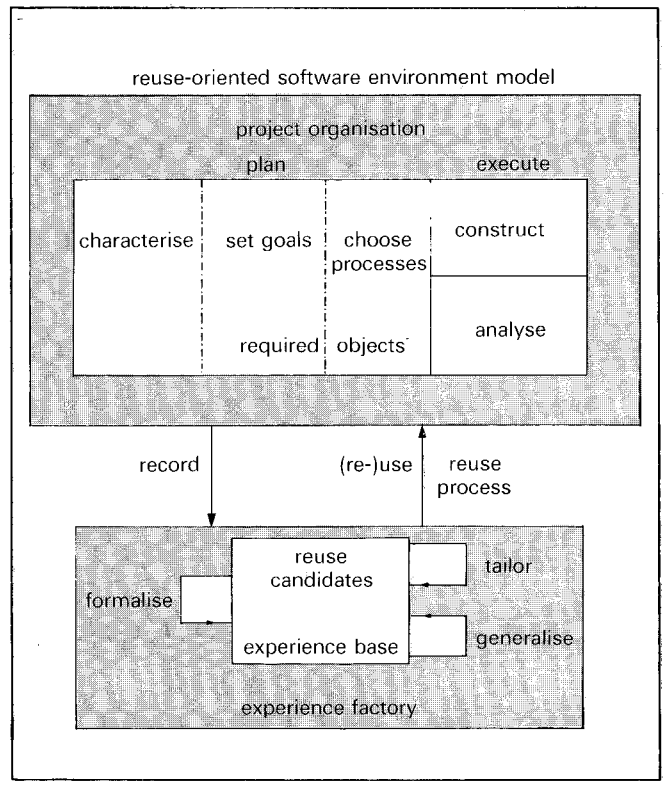

Fig. 6 Reuse-oriented software environment model

The experience base is part of an active organisational entity, referred to as the experience factory [27], which supports project developments by analysing and synthesising all kinds of experience, acting as a repository for such experience and supplying that experience to various projects on demand. In the context of the reuse-oriented software

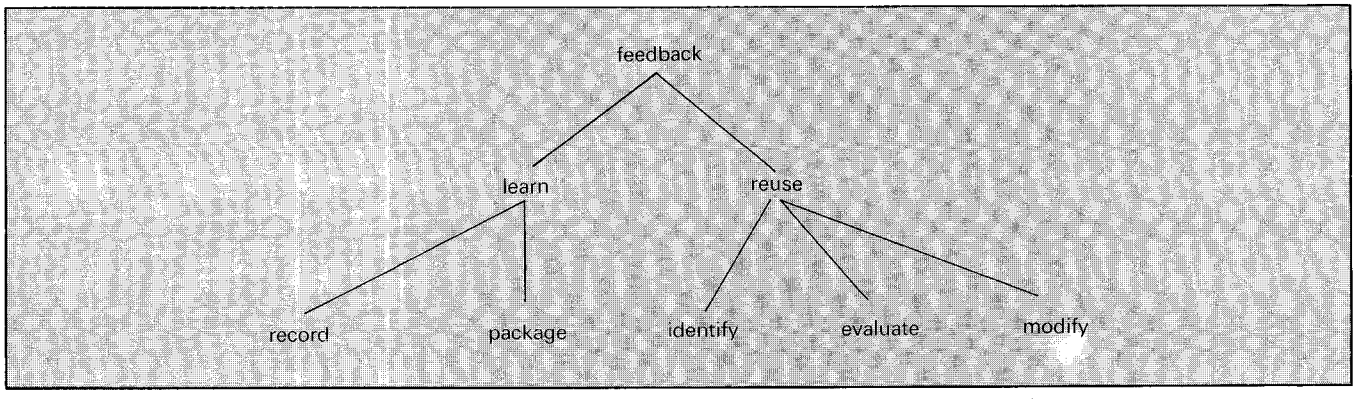

Fig. 7 Mechanisms required to support effective feedback of experience

Table 1 Characterisations of reuse candidates

Dimensions	Reuse examples		
	Ada generics	Design inspections	Cost models
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 parameters (type and number)	specification and design document required, 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 software for satellites	ground support software for satellites	ground support software for satellites
solution domain	waterfall (Ada) life-cycle model, functional decomposition design method	waterfall Ada) life-cycle model, standard set of methods	waterfall (Ada) 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. <10% prediction error)

environment model, the experience factory not only stores experience in a variety of experience bases, but also performs the constant modification of experience to increase its reuse potential. Example modifications address the formalisation of experience (e.g. building a cost model empirically based upon the data available), tailoring of experience to fit the requirements of a specific project (e.g. instantiating an Ada package from a generic package) and the generalising of experience to be applicable across project classes (e.g. developing a generic package from a specific package). It plays the role of an organisational 'server' aimed at satisfying project specific reuse requests effectively [27]. The constant collection of measurement data regarding reuse requirements and the reuse processes themselves enable the judgments needed to populate the experience base effectively and select the best-suited reuse candidates. The use of the quality improvement paradigm within the project organisation enables the integration of measurement-based analysis and construction.

5.2 Mechanisms to support effective reuse in the TAME environment model

Improvement in the reuse-oriented TAME environment model of Fig. 6 is based on the feedback of experience, captured from previous projects, into ongoing and future software developments. The mechanisms needed to support effective feedback are listed in Fig. 7.

Feedback requires learning and reuse. Although learning and reuse are possible in any environment, we are interested in addressing and supporting them explicitly and systematically. Systematic learning requires support for the recording of experience in an experience base and its packaging, in order to increase its reuse potential for anticipated reuse requirements in future developments. Systematic reuse requires support for the identification of candidate experience, its evaluation and modification.

Reuse and learning are possible in any environment. However, we want learning and reuse to be explicitly planned, not implicit or coincidental. In the reuse-oriented software development environment, learning and reuse are explicitly modelled and become desired software development characteristics. They are specific processes performed in conjunction with the experience factory.

5.2.1 Recording of experience: the objective of recording experience is to create a repository of well specified and organised experience. This requires a precise characterisation of the reuse candidates to be recorded, the design and implementation of a comprehensive experience base, and effective mechanisms for collecting, qualifying, storing and retrieving experience. The characterisation of reuse candidates is derived from characterisations of known reuse requirements and reuse processes. The characterisation of reuse candidates describes what information needs to be stored, in addition to the objects themselves, in order to make them reusable, and how it should be packaged. The experience base replaces the project database of traditional environment models. It is intended to capture the entire body of experience recorded during the planning and execution steps of software projects within an organisation.

Examples of recording experience include the storing of Ada generics, design inspection methods and cost models. Based on our reuse model, Table 1 describes the information needed, in order to make each of these object types likely reuse candidates to satisfy the hypothetical reuse requirements using the hypothetical reuse processes described in Tables 2 and 3, respectively. For example, in the case of Ada generics, we may require each object to be augmented with information on the number of instantiation parameters, the application and solution domain, and the expected or demonstrated reliability. If we can quantify such information (e.g. Ada generics developed within ground support software projects, Ada generics with less

Table 2 Characterisations of required objects

Dimensions	Ada generics	Reuse examples: Design inspections	Reuse examples: Cost models
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 parameters (type and number)	specification and design document required, 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 software for satellites	ground support software for satellites	ground support software for satellites
solution domain	waterfall (Ad) life-cycle model, object-oriented design method	Cleanroom (Fortran) development model, step-wise 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. <5% prediction error)

than five instantiation parameters are acceptable), we can use it to exclude inappropriate objects from being recorded in the first place.

5.2.2 Packaging of experience: the objective of packaging experience is to increase its reuse potential. This requires a precise characterisation of the new reuse requirements or processes, and effective mechanisms for tailoring, generalising and formalising experience. Packaging may take place at the time of the first recording experience into the experience base, or at any later time when new reuse requirements become known or our understanding of the interrelationship between reuse candidates, reuse requirements and reuse processes changes.

The objective of generalising existing experience before its reuse is to make a candidate reuse object useful in a larger set of potential target applications. The objective of tailoring existing experience before potential reuse is to fine-tune a candidate reuse object to fit a specific task or exhibit special attributes, such as size or performance. The objective of formalising existing experience before actual reuse is to increase the reuse potential of reuse candidates, by encoding them in more precise, better understood ways. These activities require a well documented, catalogued and cate-

Table 3 Characterisations of reuse processes

Dimensions	Ada generics	Reuse examples: Design inspections	Reuse examples: Cost models
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	parameterised (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 required, knowledge of system configuration plan	performed during planning stage, knowledge of project plan	performed during planning stage, knowledge of historical Ada project profiles
experience transfer	automated	human and experience base	experience base
reuse quality	correctness	predictability of modification cost	efficiency

gorised set of reuse candidates, mechanisms that support the modification process, and an understanding of the potential reuse requirements. Generalisation and tailoring are specifically concerned with changing the application and solution domain characteristics of reuse candidates; from project-specific to domain-specific to general, and *vice versa*. Objectives and characteristics are different from project to project, and even more so from environment to environment. We cannot reuse past experience without modifying it to the requirements of the current project. The stability of the environment in which reuse takes place, as well as the origination of the experience, determine the amount of tailoring required. Formalisation activities are concerned with movement across the boundaries of the representation dimension within the experience base; from informal to schematised and then to formal.

Examples of tailoring experience include the instantiation of a set of specific Ada packages from a generic package available in an object-oriented experience base; the fine-tuning of a cost model to the specific characteristics of a class of projects; and the adjustment of a design inspection method to focus on the class of defects common to the application. Examples of generalising experience include the creation of a generic Ada package from a set of specific Ada packages; the creation of a general cost model from a set of domain-specific cost models; and the definition of an application and solution domain-specific design inspection method, based on the experience with design inspections in a number of specific projects. Examples of formalisation include the writing of functional specifications for generic Ada packages; providing automated support for checking adherence to entry and exit criteria of a design inspection method; and building a cost model empirically based on the data available in an experience base.

A misunderstanding of the importance of tailoring exists in many organisations. These organisations have specific development guidebooks, which are of limited value because they 'are written for some ideal project' that 'has nothing in common with the current project and, therefore, do not apply'. All guidebooks (including standards such as DOD-STD-2167) are general and need to be tailored to each project in order to be effective.

5.2.3 Identification of candidate experience: the objective of identifying candidate experience is to find a set of objects with the potential to satisfy project-specific reuse requirements. This requires a precise characterisation of the reuse requirements; some organisational scheme for the reuse candidates available in the experience base; and an effective mechanism for matching characteristics of the project-specific reuse requirements against the experience base [28].

Let us assume, for example, that we need an Ada package which implements a 'string_buffer' with high 'reliability and performance' characteristics. This requirement may have been established during the project planning phase, based on domain analysis, or during the design or coding stages. We identify candidate objects based on a subset of the object-related characteristics stated in Table 2: 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. 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 assume that the set of successfully identified reuse candidates contains 'buffer.ada', the object characterised in Table 1.

5.2.4 Evaluation of experience: the objectives of evaluating experience are to characterise the degree of discrepancies between a given set of reuse requirements (see Table 2) and some identified reuse candidate (Table 1), and to predict the cost of bridging the gap between reuse candidates and reuse requirements. The first type of evaluation goal can be achieved by capturing detailed information about reuse candidates and reuse requirements, according to the dimensions of the presented characterisation scheme. The second goal requires the inclusion of data characterising the reuse process itself and past experience about similar reuse activities. Effective evaluation requires precise characterisation of reuse requirements, reuse processes and reuse candidates, knowledge about their relationships, and effective mechanisms for measurement.

The knowledge regarding the interrelationship between reuse requirements, processes and candidates is the result of the proposed evolutionary learning which takes place within the reuse-oriented TAME environment model. The mechanisms used for effective measurement are based on the goal/question/metric paradigm [15, 22, 26]. 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 characterisations like the ones depicted in Tables 1–3, provides very detailed guidance as to what exact metrics need to be used. For example, evaluation of the Ada generic example suggests metrics to characterise discrepancies between the reuse requirements and all available reuse candidates in terms of function, use, type, granularity and representation on a nominal scale defined by the respective categories; number of input/output instantation parameters on an ordinal scale; application and solution domains on nominal scales; and qualities such as performance, based on benchmark tests.

For example, we want to evaluate the reuse potential of the object 'buffer.ada' identified above. 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 the beginning. Three characteristics of the chosen reuse candidate deviate from the expected ones; it is more general than necessary (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 behaviour (see object quality dimension). The functional discrepancy requires instantiating object 'buffer.ada' for data type 'string'. The cost of this modification is extremely low, as the generic instantiation mechanism in Ada can be used for modification (see Table 3). The remaining two discrepancies cannot be evaluated based on the information available through the characterisations in Section 4.3. 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 behaviour. It is noteworthy that very often practical reuse seems to fail because of a lack of appropriate information to evaluate the reuse implications *a priori*, rather than because of technical infeasibility [29].

The characterisation of both reuse candidates and requirements and the reuse process allow us to understand some of the implications and risks associated with discrepancies between identified reuse candidates and reuse requirements. 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, we 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 proces, we may not have selected object 'buffer.ada' at all. If every object in our repository had performance data attached to it, we at least would be able to establish the fact that there is a discrepancy. In order to avoid the second type of problem, we need to have some (semi-) automated modification mechanism or at least historical data about the cost involved in previous similar situations. It is clear that, in our example, any functional discrepancy within the scope of the instantiation parameters is easy to bridge because of the availability of a completely automated modification mechanism (i.e. generic instantiation in Ada). Any functional discrepancy that cannot be bridged through this mechanism poses a larger and possibly unpredictable risk. Whether it is more costly to redesign 'buffer.ada' in order to adhere to object-oriented design principles or to redevelop it from the very beginning is not obvious without past experience. A mechanism for modelling all kinds of experience is given in Reference 30.

5.2.5 Modification of experience: the objective of modifying experience is to bridge the gap between a selected reuse candidate and given reuse requirements. This requires a precise characterisation of the reuse requirements, and effective mechanisms for modification. Technically, modification mechanisms are very similar to the tailoring (and generalisation) mechanisms introduced for packaging experience. Tailoring here is different, in that during modification the target is described by concrete, project-specific reuse requirements, whereas during packaging the target is typically imprecise, in that it reflects anticipated reuse requirements in a class of future projects. We refer to tailoring (and generalising) as 'off-line' (during packaging) or 'on-line' (during modification), depending on whether it takes place before or as part of a concrete instance of reuse.

Examples of modifying experience (similar to the example given earlier for tailoring) include the instantiation of a set of specific Ada packages from a generic package available in an object-oriented experience base; the fine-tuning of a cost model to the specific characteristics of a class of projects; and the adjustment of a design inspection method to focus on the class of defects common to the application.

5.3 TAME environment prototypes

In the TAME project, we investigate fundamental issues related to the reuse- (or improvement-) oriented software environment model of Fig. 6 and build a series of (partial) research prototype versions [14, 15, 29].

Current research topics include the formalisation of the goal/question/metric paradigm for effective software measurement and evaluation; the development of formalisms for representing software engineering experience, such as quality models, lessons learnt, process models, product models; the development of models for packaging experience in the experience base; and the development of effective mechanisms to support learning and reuse within the experience factory (e.g. qualification, formalisation, tailoring, generalisation, synthesis). In addition, various slices of an evolving TAME environment are being prototyped in order to study the definition and integration of different concepts.

Aspects of the TAME research prototypes, currently being developed at the University of Maryland, can be best classified by the different classes of experience they attempt to generate, maintain and reuse:

☐ support for identifying objects by browsing through projects, goals and processes based on a facet-based characterisation mechanism.

☐ support for the generalisation, tailoring and integration of a variety experience types, based on an object-oriented experience base model.

☐ support for the definition of environment-specific cost and resource allocation models and their tailoring, generalisation and formalisation, based on project experience.

☐ support for the definition of test techniques, in terms of entry and exit criteria, that provides a method for selecting the appropriate technique for each project phase, based on environment characteristics, data models and project goals.

☐ support for the definition of process models and their formalisation, generalisation and tailoring based on project experience.

☐ support for an experience factory architecture that supports the evolution of the organisation.

6 Conclusions

We have introduced a comprehensive reuse framework, consisting of reuse models, model-based characterisation schemes, the TAME environment model supporting the integration of reuse into software development, and ongoing research and development efforts toward a TAME environment prototype.

The presented reuse model and related model-based characterisation schemes have advantages over existing models and schemes in that they

- allow us to capture the reuse of any type of experience.
- address reuse candidates and reuse requirements, as well as the reuse process itself.

- provide a rationale for the chosen characterising dimensions.

We have demonstrated the advantages of such a comprehensive reuse model and related schemes by applying them to the characterisation of example reuse scenarios. In particular, their usefulness for defining and motivating the support mechanisms for comprehensive reuse and learning were stressed.

Finally, we introduced the TAME environment model, which supports the integration of reuse into software developments. Several partial instantiations of the TAME environment model, currently being developed at the University of Maryland, have been mentioned. In order to make reuse a reality, more research is required towards understanding and conceptualising activities and aspects related to reuse, learning and experience factory technology.

7 Acknowledgments

The authors would like to thank all their colleagues and graduate students who contributed to this paper, especially all members of the TAME, CARE and LASER projects; the Guest Editors, Nazim H. Madhavji and Wilhelm Schäfer; and the referees for their excellent suggestions for improving this paper.

Research for this study was supported in part by NASA grant NSG-5123, ONR grant NOOO14-87-K-0307 and Airmics grant 19K-CN983-C to the University of Maryland.

8 References

[1] BASILI, V.R., and ROMBACH, H.D.: 'Towards a comprehensive framework for reuse: a reuse-enabling software evolution environment (part I)/model-based reuse characterization schemes (part II)'. Technical Reports CS-TR-2158/CS-TR-2446 Department of Computer Science, University of Maryland, College Park, Maryland, December 1988/April 1990

[2] BASILI, V.R., and SHAW, M.: 'Scope of software reuse'. White paper, Working Group on 'Scope of Software Reuse', Proc. Tenth Minnowbrook Workshop on Software Reuse, Blue Mountain Lake, New York, July 1987

[3] BIGGERSTAFF, T.: 'Reusability framework, assessment, and directions', *IEEE Softw.*, 1987, **4,** (2), pp. 41–49

[4] FREEMAN, P.: 'Reusable software engineering: concepts and research directions'. Proc. Workshop on Reusability, September 1983, pp. 63–76

[5] PRIETO-DIAZ, R., and FREEMAN, P.: 'Classifying software for reusability', *IEEE Softw.*, 1987, **4,** (1), pp. 6–16

[6] *IEEE Softw.*, Special issue on 'Reusing software', 1987, **4,** (1)

[7] *IEEE Softw.*, Special issue on 'Tools: making reuse a reality', 1987, **4,** (7)

[8] SHAW, M.: 'Purposes and varieties of software reuse'. Proc. Tenth Minnowbrook Workshop on Software Reuse, Blue Mountain Lake, New York, July 1987

[9] STANDISH, T.A.: 'An essay on software reuse', *IEEE Trans.*, 1984, **SE-10,** (5), pp. 494–497

[10] TRACZ, W.: IEEE Tutorial on 'software reuse: emerging technology'. Catalog Number EHO278-2, 1988

[11] BASILI, V.R.: 'Can we measure software technology: lessons learned from eight years of trying'. Proc. Tenth Annual Software Engineering Workshop, NASA Goddard Space Flight Center, Greenbelt, Maryland, December 1985

[12] BASILI, V.R.: 'Viewing maintenance as reuse oriented software development', *IEEE Softw.*, 1990, **7,** (1), pp. 19–25

[13] BASILI, V.R., and ROMBACH, H.D.: 'Tailoring the software process to project goals and environments'. Proc. Ninth Int. Conf. on Software Engineering, Monterey, California, 30th March–2nd April 1987, pp. 345–357

[14] BASILI, V.R., and ROMBACH, H.D.: 'TAME: integrating measurements into software environments'. Technical Report TR-1764 Department of Computer Science, University of Maryland, College Park, Maryland, June 1987

[15] BASILI, V.R., and ROMBACH, H.D.: 'The TAME project: towards improvement oriented software environments', *IEEE Trans.*, 1988, **SE-14,** (6), pp. 758–773

[16] McGARRY, F.E.: 'Recent SEL studies'. Proc. Tenth Annual Software Engineering Workshop, NASA Goddard Space Flight Center, Greenbelt, Maryland, December 1985

[17] ZELKOWITZ, M.V. (Ed.). Proc. University of Maryland Workshop on Requirements for a Software Engineering Environment, Greenbelt, Maryland, May 1986, (Ablex Publ., 1988)

[18] CARDENAS, S., and ZELKOWITZ, M.V.: 'Evaluation criteria for functional specifications'. Proc. 12th IEEE Int. Conf. on Software Engineering, Nice, France, 26th–30th March 1990, pp. 26–33

[19] BARNES, B.H., and BOLLINGER, T.B.: 'Making reuse cost-effective', *IEEE Softw.*, 1991, **8,** (1), pp. 13–24

[20] GREEN, S., KOUCHAKDJIAN, A., BASILI, V.R., and WEIDOW, D.: 'The Cleanroom case study in the software engineering laboratory: project description and early analysis'. Technical Report SEL-90-002, NASA Goddard Space Flight Center, Greenbelt MD 20771, March 1990

[21] SELBY, R.W., Jr., BASILI, V.R., and BAKER, T.: 'CLEANROOM software development: an empirical evaluation', *IEEE Trans.*, 1987, **SE-13,** (9), pp. 1027–1037

[22] BASILI, V.R., and SELBY, R.W.: 'Comparing the effectiveness of software testing strategies', *IEEE Trans.*, 1987, **SE-13,** (12), pp. 1278–1296

[23] MILI, A., XIAO-YANG, W., and QING, Y.: 'Specification methodology: an integrated relational approach', *Softw. — Pract. Exp.*, 1986, **16,** (11), pp. 1003–1030

[24] JONES, G.A., and PRIETO-DIAZ, R.: 'Building and managing software libraries'. Proc. Compsac '88, Chicago, 5th–7th October 1988, pp. 228–236

[25] BASILI, V.R.: 'Quantitative evaluation of software methodology'. Technical Report TR-1519, Department of Computer Science, University of Maryland, College Park, Maryland, July 1985 (Proc. First Pan Pacific Computer Conf., Australia, September 1986)

[26] BASILI, V.R., and WEISS, D.M.: 'A methodology for collecting valid software engineering data', *IEEE Trans.*, 1984, **SE-10,** (3), pp. 728–738

[27] BASILI, V.R.: 'Software development: a paradigm for the future'. Proc. 13th Annual Int. Computer Software & Applications Conf., Orlando, Floria, 20th–22nd September 1989

[28] STRAUB, P.A., and OSTERTAG, E.J.: 'EDF: a formalism for describing and reusing software experience'. Proc. Int. Symp. on Software Reliability Engineering, Austin, Texas, May 1991

[29] CALDIERA, G., and BASILI, V.R.: 'Identifying and qualifying reusable software components', *Computer*, 1991, **24,** (2), pp. 61–70

[30] BASILI, V.R., CALDIERA, G., and CANTONE, G.: 'A reference architecture for the component factory'. Technical Report TR-2607, Department of Computer Science, University of Maryland, College Park, Maryland, February 1991

The authors are with the Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, USA.

The paper was first received on 29th May 1990 and in revised form on 6th February 1991.