A Framework for Evaluating Software Environments that Support Design Reuse

Hossein Saiedian and Mansour Zand
Department of Computer Science, University of Nebraska at Omaha

A practical software reuse environment must provide user-friendly facilities for the creation, collection, storage, retrieval, modification, and representation of information for reuse purposes. To evaluate the capabilities and features of such a practical reuse environment, a "checklist," "benchmark," or a "framework" is needed for evaluation purposes. We introduce such a framework. This framework includes ten properties or features that we believe are essential for any design-reuse environment. We use the framework to assess the capabilities of three software reuse environments developed at major European and North American institutions. Further research ideas related to reusing other design components are also identified.

Keywords: Design Reuse, Reuse Environments, HyperCASE, ITHACA, REBOOT

1. Introduction

Software reuse is the reapplication of a variety of components of existing system to a new system to reduce the effort of the development and maintenance of the new systems. Software reuse has become a significant subfield of research and the practice of software engineering. Reuse practice appears to exhibit considerable potential, far more than many other on-going activities, to enhance the software development process and to re-structure not only the process of software construction, but also the actual software development departments [22].

A large body of recent work consists of promoting a comprehensive methodology that utilizes sophisticated tools to produce reusable components. There is a good deal of emphasis on domain analysis as the infrastructure for reuse [1]; domain-specific software architecture to describe the topology for software components, their interfaces, and the associated computational models [18]; and domain-specific kits to produce a family of components for a specific domain. The use of knowledge representation tools and the application of CASE tools are also emphasized [9].

It is obvious that there is no single "reuse" approach as a solution for all cases. A study of the reported accounts of the relative success of software reuse in organizations such as NEC, GTE, IBM, Fujitsu, and Toshiba, in some of the DoD, NASA, and European ESPRIT III projects, and others, indicates that the major factors of success at these organization include: business modeling, organizational design, technology transfer, technology infrastructure, and the development of a standard methodology [17].

The organization of this article is as follows: A brief definition of reuse will be provided in Section 2, along with a description of its potential benefits. In Section 2.3, we will introduce the basic properties of our proposed framework. A brief description of three selected software reuse environments will be presented in Section 3. In Section 4, we evaluate the characteristics of the selected reuse environments using our proposed framework. Our summary, conclusions, and further research areas are presented in Section 5.

2. Reuse Definition

Software reuse is the use of existing software development artifacts, created during different software development projects, to create a new software product. In its purest and most productive form, reuse would start at the earliest phases of software development and continue throughout the entire process.

Design reuse is the subset of reuse where products of the design phase, both intermediate and final, are reused. Design reuse provides many advantages over code reuse. Code is very specific, and is filled with implementation details that limit its reuseability. Often these details must be modified or abstracted out of the code before it can be reused. Designs, on the other hand, have little or no implementation details, and thus can be reused for a wider range of solutions. The complete design information shows the underlying architecture of the product. It also shows the thought process of the designer, the alternatives evaluated and the reasons for selecting or rejecting those alternatives. Clearly this higher level information is more useful in a variety of software development scenarios, including new product development as well as the maintenance and re-engineering of existing products.

Karakostas [11] describes the relative benefit of reuse with this equation:

$$\text{Relative Reuse Benefit} = \frac{\text{Added Value of Reused Product}}{\text{Effort to Reuse}}$$

Clearly if the effort to reuse is small (i.e. few changes required), the benefit of reuse will be higher. Since designs contain little or no implementation details, there would be fewer changes required to reuse, thus producing a higher Relative Reuse Benefit.

Before this design information can be reused, an environment must be created to facilitate the collection, representation, storage, retrieval and modification of this information. The environment must ensure the collection of both the input to the design process, and the output. A number of such environments have been proposed. In order to evaluate these environments and assess their potential benefits, we must have a framework for evaluation purposes. That is the goal of our article: to propose such a framework and to use it in the evaluation of a number of highly practical design reuse environments.

2.1. The Software Reuse Process

The reuse processes can be classified in three categories:

- *accidental* reuse is the reuse of existing components,
- *planned* reuse is the development of reusable components, and
- *hybrid* approach which combines the first two approaches.

The third approach is also known as *domain engineering*. We discuss the third approach in this section.

The reuse process of the domain engineering approach can be built around any variation of a viable software life cycle. This process starts with an analysis of domain of the project to identify all objects or entities that are common within a given domain. The domain engineer also collects all applications in the domain and will use the information to develop a model that includes objects, operations, and relationships common to the working domain. The model and collected components (also called *assets*) are stored in a domain repository to be used for future development.

One of the major products of domain engineering is a *domain architecture* that represents the main components of a generic system. The generic domain architecture captures the principal components present in the model. To specialize the implementation of the model, the generic architecture is used and the required variations are incorporated. In general, the development process should include the following activities:

1. Identification of reusable components
2. Identification of variation
3. Specialization and generalization of the components, and
4. Development guidelines to adapt components for enhanced applications and to conform with the architectural design.

The development phases of reuse process are interleaving and might be repeated as many times as needed; different phases might take place in parallel. During the development, all products should be assessed for the degree of reuseability and be stored in the repository along with proper reuseability attributes as well as other information needed for future retrieval [2, 16].

2.2. The Impact of Reuse on Design Activities

Although some general principles have been proposed for the design of reusable components, no comprehensive methodology yet exists. In general, during the early stages of the design process, while no decisions have been made on selecting specific design alternatives, the process should capture all design components that are viable for reuseability and investigate the incorporation of existing reusable design artifacts. If the reuse process is *accidental*, all potential reusable components should be evaluated for their adaptability. The additional tasks in the design phase can be summarized as refining the requirements to search, identify, and understand the component and to select one and adapt it.

If the reuse process is *planned*, the specification of the potential reusable components must be studied and analyzed to capture their common and specific requirements and features. The outcome of this task is used to identify the potential "reuser" in the developing system as well as future development projects.

In the case of *hybrid* reuse, alternative solutions must be evaluated along with their costs, times, and their risks. The best one should be used in the design. If the domain engineering approach is used, most of the necessary bits and pieces to examine alternatives are available in a domain repository.

Making components reusable often means adding complexity to the interface, hence architectural design. During the architectural design phase, measures should be taken to ensure that the architecture is reusable; and if any component is reused, it is properly integrated into the product. As mentioned previously, the use of domain architecture may have a drastic impact on the architectural design. Once a reusable domain architecture is identified, the task of the designer is to develop a specification and design based on the generic architecture. Normally a prototype is used to test and evaluate the design and specification.

During the detailed design phase, measures should be taken to ensure that the detailed design maintains the reusable characteristics of architectural design. Also, standards and procedures for development of reusable components should be strictly followed and monitored.

2.3. A Framework to Evaluate Reuse Environments

A legitimate design reuse environment must facilitate the creation, collection, storage, representation, retrieval, and modification of the necessary information for reuse purposes. In order to more effectively and systematically evaluate such an environment, we must have a checklist that highlights the necessary features an environment must possess. For the purposes of this article, we call such a checklist a *framework* for evaluation. The term framework in this context should not be confused with its other uses in software reuse literature. We are using it as a convenient term to refer to a related collection of features, attributes, and properties. Our proposed framework consists of the features and/or properties shown in Table 1.[1]

We will briefly describe these properties as we begin to evaluate the three environments described in Section 3. Perhaps we should note that there is no theoretical foundation for the properties listed in our proposed framework. In fact, we do not yet believe such properties can be theorized in the near future, because there is no formal theory behind software reuse methodologies. Every heuristics reported is based on experiences, observations, and intuition. To some extent, reuse concepts are similar to their counterparts in object-oriented field. While there is not (yet) a formal theory of object-orientation, it is agreed that applying object-oriented concepts will improve software engineering practices and will yield better software. Similarly,

[1] It should be made clear these features are purely technical features. In general, when an organization plans to purchase a tool, other factors such as price, compatibility with existing tools and operating systems, etc., play a role. However, our goal here is to focus on and evaluate technical factors.

No	Property
1	Design methodology independent
2	Support for object-oriented techniques
3	Representation of Design Properties
4	Support for Automated Tools
5	Support for Traceability
6	Hypertext Capabilities
7	Version Control Features
8	Implementation-Language Independence
9	Partitioning by Domain
10	Automated Consistency Checking

Table 1. Framework Properties

the properties proposed in Table 1 are those that we believe an environment must have in order to allow for effective software reuse. We have collected these properties through our practical and research work, our observation of certain environments, and our study of reuse literature such as [1], [3], [5], [8], [10], [15], [19], [20].

3. Description of the Selected Reuse Environment

As indicated earlier, an environment must be created in order to effectively use design information for reuse. Such an environment must facilitate the collection, representation, storage, retrieval, and modification of design information for practical purposes. We have selected three such reuse design environments. These three environments were selected based on the following criteria:

1. The selected environment had to be already developed.
2. The selected environment for design reuse had to be (a part of) a complete environment that supported full life-cycle reuse.
3. The selected environment had to support the majority of the characteristics described in our proposed framework.

Perhaps many environments have the potential to satisfy the above criteria. We chose two European and one North American reuse environments: *HyperCASE, ITHACA, and REBOOT.* There are certainly other tools and environments that provide support for reuse. The ones that are being evaluated here are just a representative selection. Developed by major organizations and/or institutions, these reuse environments have been shown to be quite effective. Before comparing and evaluating the environments themselves, we provide a brief description of each.

3.1. HyperCASE

HyperCASE [5] is a software engineering environment being developed by the Amdahl Australian Intelligent Tools Program. As its name implies, the environment was developed on top of an extended hypertext system. The goal of HyperCASE was to develop an integrated CASE environment, where a number of standalone CASE tools could be merged into this environment and work together as one. The integration occurs by combining a hypertext based user interface with a common knowledge-based document repository.

The system architecture can be seen in Figure 2. The HyperEdit portion of the system is the hypertext based user interface. Among other things, it provides access to a number of diagramming tools and text editors to facilitate the joining and modification of reusable artifacts. The HyperBase portion of the system provides all the tools necessary for all phases of software development. The Base tools are provided to manage the underlying hypertext structures. The CASE tools exist to build solutions and to ensure that the solutions are consistent with the knowledge-base and the design solutions are reusable. HyperDict is the final piece

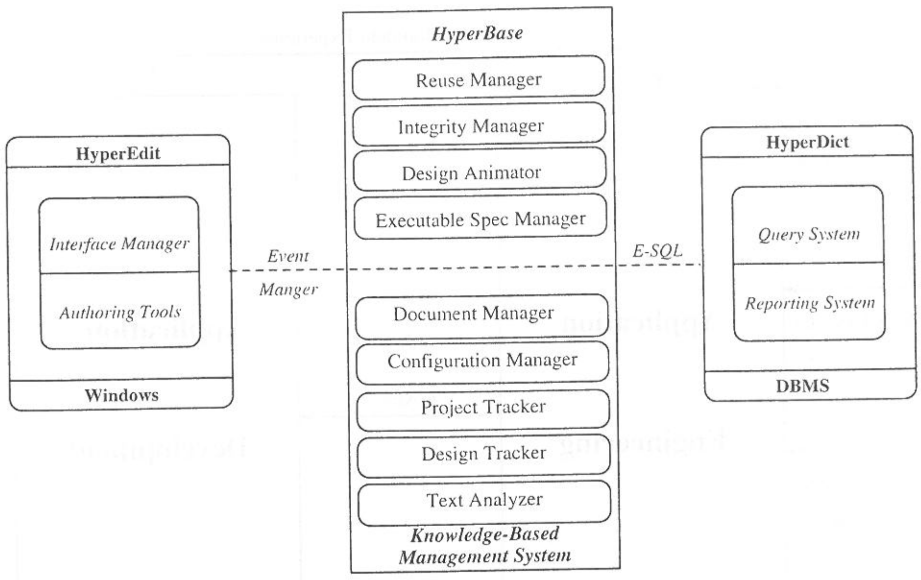

Fig. 1. HyperCASE System Architecture

of the HyperCASE environment. It is the actual repository for all documents stored in HyperCASE. It is implemented as a Prolog data store in Ingres.

The framework for design reuse in HyperCASE is constructed and maintained by three of the tools within HyperBase, the design tracker, the design animator and the reuse manager. The reuse manager is responsible for creating and maintaining a uniform declarative representation of all documents, both design related and otherwise. It uses this standard representation to formulate a set of knowledge-based rules about the design to aid in its reusability. In addition, it provides a means for multiple classifications of design artifacts. The design tracker is the tool used to capture all the steps involved in the design process. It forces designers to record all their decisions and the reasons why. When reusing a design artifact, the design tracker will allow the designer to retrace the original design process step-by-step, thus promoting a good understanding of the artifact. Finally, the design animator is responsible for monitoring all modifications and refinements made during a software development. Its job is to ensure that there is complete traceability, both structural and functional, from requirement specification all the way to coding. The design animator will allow a developer to execute a completed module in a special debug mode. In this mode, the executing program can be suspended at any time. Once suspended, the developer can request that the design animator display any portion of the software development, from requirements to design to code, which corresponds to the point at which the program was suspended. This can be a very powerful testing tool.

3.2. ITHACA

Intelligent Tools for Highly Advanced Commercial Applications (ITHACA) [6, 8, 15, 20] is a software application development environment, created explicitly to enhance the utilization of reuse. ITHACA is being developed as part of the ESPRIT II and ESPRIT III projects. ESPRIT II and ESPRIT III (European Strategic Program for Research in Information Technology — Phase II and III) are long term efforts sponsored by the countries of Europe to develop a complete software development environment to support commercial software development today and for years to come. The ITHACA environment was designed from the ground up to support reuse of both software components and software development artifacts (i.e. requirements, specifications, designs).

The software development methodology used within ITHACA is not a commonly used methodology, and appears to be unique to ITHACA at the moment. Within ITHACA, software development is divided into two separate but complimentary phases, Application Engineering (AE) and Application Development (AD).

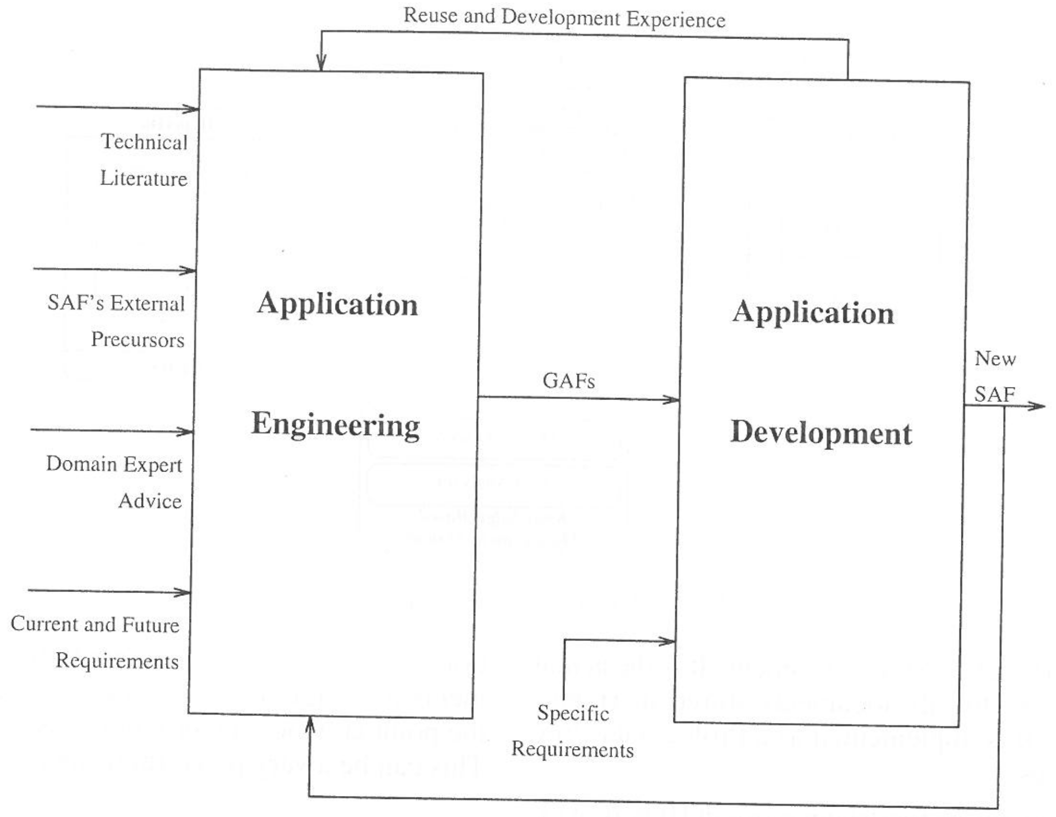

Fig. 2. ITHACA Environment

Application Engineering involves the creation of Generic Application Frames (GAFs). A GAF is an object oriented framework or skeleton for developing applications within a specific problem domain. The GAF contains classes of information which define generic development information (requirements, specifications, designs etc.), and knowledge about how to reuse this information. GAFs are normally developed by experts in both software development and the domain associated with the GAF.

The Application Development phase is completed when an actual application needs to be developed. A developer will initially obtain some high level requirements for the application. Using those requirements the developer will use ITHACA to scan all existing GAFs for a frame(s) which seems to fit the requirements. Ideally, there will always be a GAF for every possible problem domain, however, if one does not exist it will have to be created. Once the developer selects the best GAF, he/she simply uses the generic guidelines, development information and reusable artifacts as a basis for constructing the application. If additional structures or components are required, they are built by the developer and identified as new knowledge to be abstracted back into the GAF(s) associated with this domain. After all the components have been tailored and/or built, the components are assembled using a script which defines the interaction between the components, normally design classes. The newly constructed application is known as a SAF, Specified Application Frame. Any new domain knowledge discovered during SAF development is then incorporated back into the GAF for future use. A detailed diagram of this data flow can be seen in Figure 3.

The architecture of the ITHACA ADE is shown in Figure 4. The SIB or Software Information Base is a database repository of all the object classes which make up the GAFs and SAFs. On the AD side, there are a number of tools which support the developer in SAF construction. RECAST is used for selection of a GAF and refinement or specification of the generic information. VISTA is a visual scripting tool which allows the developer to link all the selected components into a single application. VISTA allows the developer to explicitly define the interaction between the components. Once

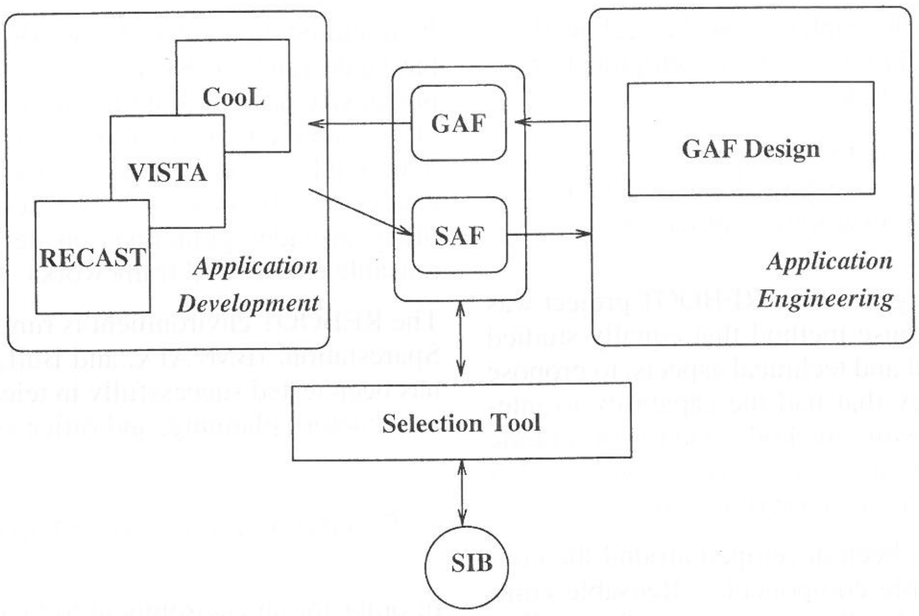

Fig. 3. ITHACA Application Development Environment

fully scripted, the application is simply generated into CooL, which is the object-oriented language used within ITHACA.

3.3. REBOOT

REBOOT (REuse Based on Object-Oriented Techniques) is a product of ESPRIT III project. More specifically, REBOOT is ESPRIT Project 5327. REBOOT is not just a tool, it is rather a collection of integrated tools and methods for the creation, adaptation and reuse of components. The REBOOT project team was made by members of ten European organizations. This project started in 1990 and was completed in 1994. The objective of the project is to improve productivity and enhance quality of software development through promoting reuse. The reboot project goes beyond the design and development of software development environment. It also provides methodologies for studying managerial, commercial and legal aspects reuse, training package and support systems [12] and [14].

The major motivation of the REBOOT project was to overcome weaknesses of previous reuse projects such as:

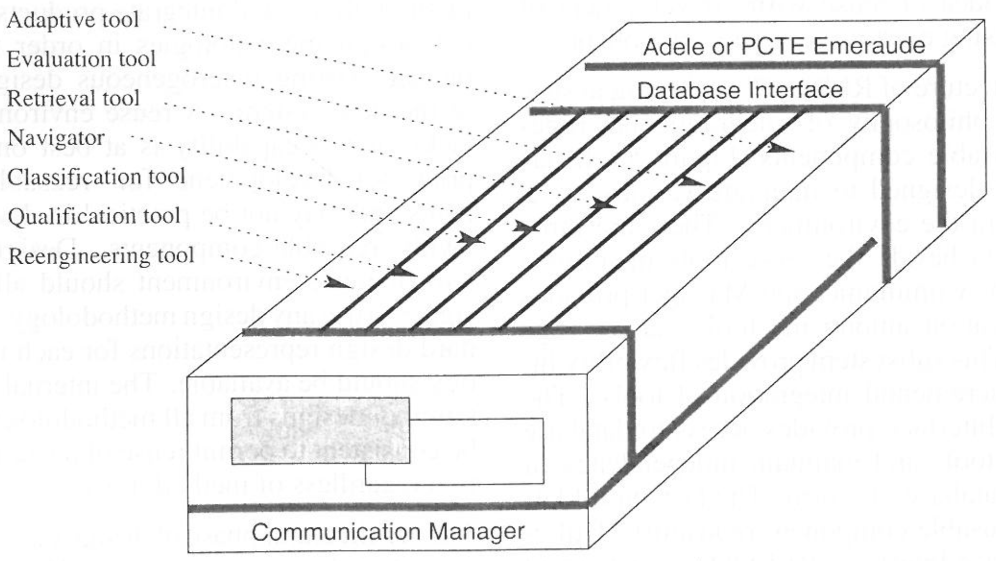

Fig. 4. Architecture of the REBOOT Environment

1. Too much emphasis on the technical aspects and focus on the development of sophisticated tools.
2. Underestimating the organizational problems and requiring dramatic changes in software development practices.

Therefore the goal of the REBOOT project was to prompt a reuse method that equally studied organizational and technical aspects, to propose a methodology that had the capability to integrate with existing methods, and to concentrate on utilization of existing technologies rather than focus on innovation of new ones [19].

REBOOT has been developed around the concept of reusable components. Reusable components include all products of software lifecycle as well as all information about the components. Component repository of REBOOT supports heterogeneous components that have been developed by different methodologies and teams.

The major components of REBOOT environment include a database (i.e. reusable components repositories) and a two groups of tools. The Component Building Assistant, the first group, is used for building, classification, and qualification of new components. This subsystem supports the idea of reuse **for** concept: development of new inherently reusable components. The Reuse Assistant is used for retrieval, adaptation, and evaluation of existing components in repositories. The Reuse Assistant supports the idea of reuse **with**: development of systems using existing reusable components.

The architecture of REBOOT environment supports the philosophy of integrating heterogeneous reusable components, Figure 5. It has also been designed to integrate various existing tools in the environment. There are three major parts beside the reuse tools mentioned above. The Communication Manager provides communication among the tools via message passing. This subsystem provides flexibility interm of incremental integration of tools. The Database Interface provides access to database for reuse tools and maintain independency of tools on database platform. The Database Platform is reusable component repository. Utilizing Database Interface, REBOOT environment can use different databases.

It should be mentioned that REBOOT does not have a design tool. However, it provides a comprehensive design guideline for reuse with any object-oriented methodology. There are two main kinds of design components in REBOOT: classes and framework. The design methodology provides guidelines on development of reusable classes and frameworks [19].

The REBOOT environment is running on Sun4 Sparcstation, IBM/AIX, and Bull/DPX20 and has been tested successfully in telecommunication, factory planning, and office systems [14].

4. Evaluating the Reuse Environments

In order for an environment to facilitate design reuse and allow the maximum benefits to be attained, it must possess characteristics such as those listed in Table 1. These characteristics form the basis for the comparison of the environments described above. The characteristics were gathered from a variety of sources as indicated by the references. This portion of the paper lists the characteristics, a brief description of each and then a summary of each of the environments' support for that characteristic.

4.1. Design Methodology Independence

The environment's design methodology must be relatively independent [21]. A reuse environment should be able to handle different design methodologies and integrate products of different design methodologies in order to assemble an existing heterogeneous design artifact in the new system. A reuse environment that lacks such a capability is at best only appropriate for development "for" reusable components and may not be practical in development "with" reusable components. Design Subsystem of reuse environment should allow a designer to use any design methodology. The standard design representations for each methodology should be available. The internal representation of designs from all methodologies should be consistent to permit reuse of design information regardless of methodology.

The architectural phase of design has significant impact on designing a solution. This step of design defines a high-level strategy for solving the problem and constructing a solution. Providing the capability to reuse different architectural artifacts as well as methodologies must be included in the future software environments.

Evaluation

HyperCASE: YES — HyperCASE will allow the designer to use any design methodology. The components of the design are simply connected via hypertext links. The standard representations for most design methodologies are available to be displayed graphically.

ITHACA: NO — ITHACA has established its own software development methodology. ITHACA's methodology can be described as *software construction.* Although ITHACA does not incorporate the standard design representations, it does provide graphical representations unique to its design methodology.

REBOOT: YES but limited — REBOOT mainly relies on an object-oriented approach, but not on a specific object-oriented methodology. The REBOOT design guidelines for reuse can be applied on any object-oriented methodology (and to some extent only with other types of methodologies).

4.2. Support for Object-Oriented Technique

Object-Oriented techniques have become one of the major techniques in software development. Although object-oriented techniques by themselves do not provide systematic reuseability, the concepts incorporated in this methodology and components such as classes and objects can facilitate this goal. Therefore, it is important that the environment supports object-oriented techniques, among other methodologies.

In an object-oriented environment, it is important that the environment internals be implemented via object-oriented techniques [15, 20]. The internal representation of the design information should be stored as objects. The environment should make use of object-oriented techniques whenever possible to promote the logical grouping of design artifacts by real world entities.

Evaluation

HyperCASE: Limited — The HyperDict is a relational database, and therefore is not purely object-oriented. However, the Event Talk or Event Manager is able to provide the user of HyperEdit an object-oriented view of the repository.

ITHACA: YES — Each piece of knowledge associated with each GAF and SAF is stored as a class in the SIB. Every aspect of ITHACA was developed using object-oriented techniques, and these techniques pervade the entire system.

REBOOT: Indirectly — REBOOT is not a design tool. It only manipulates files generated by the user's design tool. These files are managed as objects with attributes and relationships that respectively record the characteristics of the design and its dependence on other related objects.

4.3. Representation of Design Properties

The environment should allow representation of the design related properties. [4]. These properties are commonly given as: (1) Architectural design. The architecture of the design must be available, and be able to be generated into the standard architectural design representations (e.g., a DFD). (2) Detailed design. The detailed design must be available, and be able to be generated into the standard detailed design representations. (3) Design decisions and justification. Each design decision that was made must be recorded along with a justification for the decision. This will allow a future maintainer or reuser of this design to easily ascertain the logic behind the design. (4) Design alternatives and reasons for rejection. It is also very important to maintain a record of the design alternatives that were considered and rejected. This will also be very beneficial to a future maintainer or reuser. It will save them from having to make the same mistakes, or learn the same lessons, as the original designer.

Evaluation

HyperCASE: YES — This information is gathered by the design tracker and is stored in the HyperDict. Standard representations for this information are supported by HyperCASE.

ITHACA: YES — This information is contained in the GAFs or is captured during the SAF development. If necessary, new information from the SAF development will be abstracted back into the associated GAF. The information is stored in the SIB.

REBOOT: This environment relies on the user's design tool (which can simply be drawing and natural language). REBOOT design guidelines strongly recommend rècording all details of design decision making, justifications and alternatives in an associated file, using a preferred word processor or text editor.

4.4. Support for Automated Tools

The environment should support automated browsing, querying and manipulation of design information. [5]. The volume of design artifacts that will be accumulated necessitates this requirement. It is imperative that a designer be able to build a query to find design information related to the problem at hand. A sequential search through all the artifacts would be all it would take to discourage any designer from trying to reuse design artifacts. The same argument holds for manipulation of design artifacts. The designer should be able to use automated tools to make any necessary changes to a design artifact that is to be reused. The system should also take care of establishing the link between the original artifact and the new modified version.

Another crucial aspect of an automated tool is its potential support for computer-supported cooperative work. Among the facilities that a groupware can provide are multi-user editors, message systems, coordinating system and so forth. This is a worthwhile tool for large-scale reuse projects. Such a tool assists group interaction in all software development projects. In reuse projects it can help re-users by passing on their experiences with components or requests for information on needed components, if they are not in the repository or the repository doesn't have necessary information.

Evaluation

HyperCASE: YES — HyperEdit supports hypertext navigation, updates and natural language queries of the artifact repository.

ITHACA: YES — RECAST provides the tools to browse the GAFs, make a selection and modify the GAF as required.

REBOOT: YES — support to browse, query reusable components, identify the design document (or design file) and display it by the appropriate tool (automatic activation of the corresponding word processor or design tool).

4.5. Support for Traceability

The environment should support traceability to adjoining lifecycle phases [4]. This requirement addresses the necessity to be able to track a particular design artifact back to its original requirement or forward to its implementation. This is outside the scope of design reuse, but is very necessary if the full benefits of reuse are to be obtained.

Evaluation

HyperCASE: YES — The Hypertext links between corresponding requirements, designs and code are built and maintained by the design animator.

ITHACA: YES — Within the GAF the generic requirements information is directly associated with the appropriate generic design solutions which are associated with the appropriate script and code. The same is true within the SAF, only to a greater, more detailed level.

REBOOT: YES — The REBOOT component model considers a Reusable Component to be an aggregate. This means an RC is composed of various workproducts (Requirements, specification, design, code, tests, documents, etc.). REBOOT also have an 'is-an-evolution-of' relationship to relate versions of the RC, but it is not yet efficiently managed.

4.6. Hypertext Capability

Flexible and powerful internal environment structure (hypertext) is considered an important feature [4, 5]. This requirement is necessary for the satisfaction of nearly every other characteristic in this list. Without a hypertext like structure, navigation through artifacts, linking of related artifacts, attaching related documentation, setting up traceability and providing version control would be very complex. With hypertext, artifacts can be grouped and logically linked together to promote maximum flexibility and utility. Besides, hypertext adds extensibility to the system internals, where no other structure could.

Evaluation

HyperCASE: YES — Hypertext is the base upon which HyperCASE functions.

ITHACA: Unknown — The details of the internal structures of the environment were not addressed.

REBOOT: YES — The Hypertext capability is provided through relationships using Navigator subsystem.

4.7. Version Control Features

Another important consideration is support for version control of designs and related information [4, 5]. Over the course of the design phase of software development, the designer will most certainly experiment with a number of approaches. Version control will allow the designer to incrementally develop the design, and always be able to retreat back to an earlier version of the design, if necessary. Such functionality is also very desirable, once the maintenance phase of software development is reached.

Evaluation

HyperCASE: YES — Version control within HyperCASE is managed by the Configuration Manager tool within HyperBase.

ITHACA: YES — Within the SIB, the class objects are version controlled. However, there was no information to indicate that a designer could explicitly create different versions of the same design and let the system manage the separation and control of each version.

REBOOT: Limited version control is available.

4.8. Implementation Language Independence

The environment should be independent from any implementation language [21]. The representation of the design within the environment should not be tied to any specific implementation language. In other words, the same detailed design should be able to be translated into any implementation language available within the environment.

Evaluation

HyperCASE: YES — HyperCASE currently provides no code generation facilities. However, since it supports all standard design methodologies, it stands to reason that the code could be generated into any language.

ITHACA: NO — ITHACA was developed with a single implementation language in mind. CooL is the object-oriented language selected for the ITHACA environment. Since CooL is such an integral part of the total environment, this factor will not impact the system negatively.

REBOOT: YES — REBOOT design presentation of object (file managed as objects) is independent from the implementation language.

4.9. Partitioning by Domain

The frame should support partitioning of design information by domains [8, 20]. It is generally agreed that reuse is most effective within a well-defined, fairly small domain. The environment should allow the designer to specify which domain to place design artifacts into. The environment should also provide limited sanity checks, to verify the selected domain is appropriate. The environment should also have the ability to select an appropriate domain if one is not specified by the designer.

Evaluation

HyperCASE: NO — HyperCASE provides no direct support for domain grouping. However, it could be implemented as a field in the HyperDict representation of all the artifacts.

ITHACA: YES — Domain partitioning is the single criteria for creating separate GAFs. GAFs are built for a specific domain. SAF development that occurs for an application in a particular domain will use only GAFs developed specifically for that domain.

REBOOT: YES — REBOOT uses a sophisticated faceted classification per domain for partitioning of domain. This is a revised facet based approach to tailor object-oriented components. The classifier subsystem is very flexible and allow the designer to specify which domain to place design artifacts into. However, it doesn't have the ability to select an appropriate domain if one is not specified by the designer.

4.10. Automated Consistency Checking

The environment should support an automated means of design solution consistency checking [13]. The representation of the design information should be done in such a way that the design solution can be verified against the corresponding requirements specifications. The characteristic extends beyond the bounds of design, but it is a very necessary characteristic. Without this type of verification the designer would either skip this step alltogether, or be forced to manually trace each part of the design back to the specifications and perform the checking. Either solution is unacceptable.

Evaluation

HyperCASE: YES — The Integrity and Completeness Manager contains the inferencing environment for checking document completeness, solution completeness and semantic integrity.

ITHACA: YES — ITHACA makes use of an object-oriented model called the *Objects with Roles Model* (ORM). As components are tailored and created, the RECAST tool verifies that the ORM model holds. As components are linked and applications built, the VISTA tool verifies the ORM model.

REBOOT: NO — REBOOT considers this as the responsibility of the design tool and not as the responsibility of the reuse tool.

4.11. A Tabular Comparison

A tabular comparison of the three reuse environments using our proposed framework is shown in Table 2.

No	Property	HyperCASE	ITHACA	REBOOT
1	Design methodology independent	Yes	No	Yes
2	Framework internals implemented using Object-Oriented techniques	*Limited*	Yes	*Indirect*
3	Allow representation of the design related information	Yes	Yes	No
4	Support automated browsing and querying of design information	Yes	Yes	Yes
5	Support traceability to adjoining life-cycle phases	Yes	Yes	Yes
6	Flexible and powerful internal framework structure	Yes	*Unknown*	Yes
7	Support version control of designs and related information	Yes	Yes	*Limited*
8	Implementation language independent	Yes	No	Yes
9	Support partitioning of design information by domain	No	Yes	Yes
10	Support an automated means of design solution consistency checking	Yes	Yes	No

Table 2. Evaluation Summary

5. Conclusions, Summary, and Further Research

Each of the three environments plainly supports design reuse as they were developed to do. The question that faces software engineers and their managers is, "Which environment better suits the needs of my shop and my developers?" Based on the comparisons just completed, the answer to that question is found in the answer to another question, "What software development methodology is right for the products we produce?"

Each of the three environments provides a unique choice of methodology. HyperCASE lets the software engineer determine the design methodology to be used. HyperCASE automatically keeps track of reusable information, and can recommend reusable artifacts to solve a problem, but it is up to the software engineer to "reuse or not to reuse". ITHACA can be viewed as an intermediate level between the other two. It has its own unique development methodology, but the decision on what to reuse and how, is left to the application developer, as in HyperCASE. Which environment is appropriate for any given software development shop can only be decided by those who work there. Here are some general guidelines.

If the development shop is fairly immature in software engineering techniques, it may be better served by a tool within the PA environment. HyperCASE takes a lot of the learning curve for software engineering techniques and puts it into the tool. For shops with good and solid software engineering practices established, but little or no reuse, ITHACA would be a good selection. ITHACA will allow the on-site experts to use their expertise in GAF design, while allowing the rest to build applications (SAFs) with greater ease, thanks to the experience built into the GAFs. For shops that require maximum flexibility, HyperCASE is the answer. HyperCASE is also better suited for shops that desire to reverse engineer existing applications to generate reusable artifacts. It is simply the most adaptable. So the decision is left where it should be, in the hands of the managers and leaders of software developers today. Those managers with a clear view of their environment and the software demands of the future will choose to implement an environment such as the three described above. Without it the software crisis will surely overwhelm them. Their products will then be taken over by a manager whose team stays one step ahead of the software crisis.

5.1. Additional Research Items

Development of a standard for design representation. This is the single greatest missing link in the design reuse effort. Currently, all efforts to reuse designs are developed independent of each other, with no common ground. Although these efforts may provide excellent support for reuse of designs, they lack the ability to export (and import) their reusable designs to other environments. It is only through the ability of all environments that advantage may be taken of the work of others, in order to obtain the required benefits of design reuse.

Development of a framework for the reuse of formal specifications. Currently, software development environments do not take full advantage of the use and reuse of formal specifications. Traditional efforts on formal specifications have been for functional specification of a software system and have largely focused on abstractions techniques (and refining abstractions into some implementation). To make formal methods an integral part of industrial software development, the use of these methods has to be as cost-effective as possible. One way of achieving such cost-effectiveness is through development of a framework for reuseability of already developed formal specifications. Such a framework has numerous benefits [7]:

- If a single specification can serve a number of products, its development cost can be amortized over those products.
- The development of several products from the same specification can lead to uniformity across those products as well as their development.
- Reuseability in specification may lead to correspondingly reusable products.
- The fact that a specification can serve as framework for several products, may cause its developers to strive for particularly elegant abstractions. This in turn may lead

to cleaner definitions of the fundamental concepts behind related applications.

- The job of defining specifications for the framework may be delegated to a small team of highly skilled engineers.
- Libraries of formally specified software components that form the basic design repertoire of software developers may gradually be produced.

Thus Research is needed to develop a framework for reusing formal specifications.

Formalization of a design methodology based on reuse. Such a methodology is under development at the Software Engineering Institute (SEI) [10]. The ITHACA frameworks also provide possible examples of this type of methodology. However, these methodologies need to be formalized so that their use can be picked up and used elsewhere.

5.2. Summary

The challenges of incorporating design reuse into an established software development organization are great. However, if there were easy answers to the software crisis, there wouldn't be a crisis. As can be seen from this paper, the tools for implementing design reuse are available today. One of the main inhibitors is that the top mangers of organizations refuse to take a long-term look at software development process. A long-term view, including the change to incorporate reuse as an integral part of the software development process, would cost more up front, and many managers are unwilling to take that risk.

It is the duty of all software professionals to educate others, and promote the advancement of new technologies within software engineering. Until this education takes place, and the people who control the purse strings are convinced of the value of reuse, the software industry is doomed to continue to wallow around in the software crisis. On the benign side, if the education does take place, the science of software development can move forward, out of the software crisis, and into the software revolution.

Acknowledgment. The authors would like to acknowledge David Barr's efforts and contributions during the earlier parts of this project. The director of the REBOOT Project, Jean-Marc Morel, provided many comments and supplied the necessary materials for evaluating the REBOOT environment.

Hossein Saiedian's work was supported by a UCR Summer Fellowship grant.

References

[1] G. Arango. Domain analysis methods. In W. Schaeffer, R. Prieto-Diaz, and M. Matsumoto, editors, *Software Reusability*, pages 17–49. Ellis Horwood, New York, 1993.

[2] G. Arango. A brief introduction to domain analysis. In *Proceedings of the ACM Symposium on Applied Computing*, pages 42–46, 1994.

[3] J. Bell. Reuse and browsing: Survey of program developers. In D. Tsichritzis, editor, *Object Frameworks*, pages 197–220. Université de Genève, 1992.

[4] T. Biggerstaff. Design recovery for maintenance and reuse. *IEEE Computer*, 22(7):36–49, July 1989.

[5] J. Cybulski and C. Reed. A hypertext based software engineering environment. *IEEE Software*, 9(2):62–68, March 1992.

[6] M. Fugini, O. Nierstrasz, and B. Pernici. Application development through reuse: the ITHACA tools environment. *ACM SIGOIS Bulletin*, 13(2):38–47, August 1992.

[7] D. Garlan and N. Delisle. Formal specifications as reusable frameworks. In *VDM'90*, LNCS 428, pages 150–163. Springer-Verlag, 1990.

[8] R. Girardi. Application engineering putting reuse to work. In D. Tsichritzis, editor, *Object Frameworks*, pages 137–149. Université de Genève, 1992.

[9] M. Griss and K. Wentzel. Hybrid domain specific kits for a flexible software factory. In *Proceedings of the ACM Symposium on Applied Computing*, pages 47–52, Phoenix, AZ, March 1994. ACM.

[10] K. Kang, S. Cohen, R. Holibaugh, J. Perry, and A. Peterson. Reuse-based software development methodology. Technical Report CMU/SEI 92-SR-4, Software Engineering Institute, Carnegie Mellon University, January 1992.

[11] V. Karakostas. Requirements for CASE tools in early software reuse. *Software Engineering Notes*, pages 39–41, April 1989.

[12] E. Karlson. *Software Reuse: A Holistic Approach.* Software Based Systems. Wiley, 1995.

[13] M. Lubars. Representing design dependencies in an issue-based style. *IEEE Software*, 8(4):81–89, July 1991.

[14] J. Morel. The REBOOT environment. Technical Report 7808, ESPRIT Project, 1994.

[15] O. Nierstrasz, S. Gibbs, and D. Tsichritzis. Component-oriented software development. *Communications of the ACM*, 35(9):160–165, September 1992.

[16] R. Prieto-Diaz. Systematic reuse: A scientific or an engineering method? In *Proceedings of the ACM Symposium on Software Reusability*, pages 9–10, 1995.

[17] W. Schaeffer, R. Prieto-Diaz, and R. Matsumoto, editors. *Software Reusability*. Ellis Horwood, 1993.

[18] M. Shaw and D. Garlan. *Software Architecture*. Prentice-Hall, 1996.

[19] G. Sindre and R. Conradi. The REBOOT approach to software reuse. *Journal of Systems and Software*, 30(3):201–212, September 1995.

[20] C. Trotta and O. Nierstrasz. Object-oriented support for generic application frames. In D. Tsichritzis, editor, *Object Frameworks*, pages 151–195. Université de Genève, 1992.

[21] A. Wasserman, P. Pircher, and R. Muller. The object-oriented structured design notation for software design representation. *IEEE Computer*, 23(3):50–63, March 1990.

[22] M. Zand and M. Samadzadeh. Software reuse: Current status and trends. *Journal of Systems and Software*, 30(3):167–170, September 1995.

Received: November, 1996
Accepted: December, 1997

Contact address:
Hossein Saiedian and Mansour Zand
Department of Computer Science
University of Nebraska at Omaha
Omaha, Nebraska 68182–0500
USA

HOSSEIN SAIEDIAN (Ph.D., 1989, Kansas State University, USA) is an associate professor at the Department of Computer Science, the University of Nebraska in Omaha, USA. He is a member of the IEEE Computer Society, Sigma Xi, the ACM, and currently serves as the Chair of the ACM SIGICE (Special Interest Group in Individual Computing Environments). Dr. Saiedian's contributions in software engineering are extensive and recognized.

MANSOUR ZAND is an associate Professor at the Department of Computer Science, the University of Nebraska in Omaha. He received his Ph.D. in Computer Science from Oklahoma State University. Dr Zand's main area of research is software engineering, specifically software reuse, about which he has published more than 35 papers. He is a co-founder of ACM-SIGSOFT Symposium on Software Reuse (SSR) and has organized several other Reuse events and forums. His most recent work on this area is related to organizational impacts of software reuse, legal issues, and fuzzy metrics. He is also involved in research in the database area, specifically data-warehousing, object oriented and distributed database systems.