ABSTRACTION-BASED ENVIRONMENTS

Grady H. Campbell, Jr.

TABLE OF CONTENTS

ABSTRACT

One of the goals of a software development environment is to increase the productivity of software developers. Generally, this has resulted in the automation of established manual methods of software development. Such an approach cannot produce adequate improvements in productivity - the complexity of the software development process must be significantly reduced. This paper proposes an approach built upon the exploitation of abstraction and software reuse to change the focus of software development. A concept of abstraction-based environments results and is illustrated by an operational version of such an environment.

1 INTRODUCTION

A software development environment (SDE) designed to automate conventional methods of software development cannot significantly enhance the productivity of developers. Such methods have evolved as a bridge across the enormous gap between the concepts of application domains and the concepts of software implementation and its tools. SDEs are needed that enhance software developer productivity by orders-of-magnitude but this requires refinements of the perceived software development process that will eliminate major sources of effort by significantly narrowing the domain/software gap. An adequate SDE must exploit the concepts of abstraction to reduce the complexity of the process and to enable substantial software reuse.

A program family is defined as a set of programs characterized by a set of common properties whose understanding will aid the understanding of individual members [Parnas]. A software development environment can be viewed as a mechanism for the construction of members of such a family wherein the constructable programs comprise exactly that family; a particular derived program is distinguished by the decisions made through the environment about that program as it is constructed. For an environment to be abstraction-based, it must embody a concise and coherent conceptual model of its associated program family. Software AE's Spectrum environment is such an environment.

The essence of an abstraction-based environment is a conceptual model of the components of an application such that each component is presented in terms of appropriate abstract description paradigms. Because such an SDE is based on a concise description of a program family, it is easier to understand and use. A family description provides a model that organizes and simplifies a user's view of programs that can be built through the environment. An instantiation of the conceptual model and the necessary set of abstract component descriptions constitutes an application system specification. This method requires a fundamental paradigm shift in how software development is approached, in that systems are essentially "described" rather than coded.

Database management systems (DBMS) illustrate the benefits of abstraction. Consider a DBMS from an SDE perspective. A database is derived as an instantiation of the DBMS data definition schema. A database family is the set of application databases that can be built using that DBMS. The data model for the DBMS is the user model of all constructable databases. The schema language is the user medium for describing a particular application database. Common capabilities of all databases are represented by the data access facilities (query language), storage mechanisms, concurrency control, and recovery capabilities implemented by the DBMS. Unique capabilities of a particular database are represented by its specified schema and choices made about its physical representation. Any database that fits the data model can be implemented via the DBMS; all other

databases are excluded. The success of DBMS technology is evidence that appropriate abstraction can lead to major application productivity gains.

The following sections describe the design principles underlying abstraction-based environments, features of the Spectrum abstraction-based environment, results and conclusions from this experience, and other related and future work.

2 DESIGN PRINCIPLES

The motivation for abstraction-based environments is a belief that traditional software development approaches overemphasize technological concerns. The first goal of an SDE should be to minimize reformulation of a solution when the problem domain is understood. Traditionally, the effort to correlate domain concepts to software components is a major development cost, particularly as user requirements change.

2.1 Environment Structure

An abstraction-based environment has three basic functions: "specification" - the incremental acquisition and presentation of system descriptions; "generation" - the derivation of a specification-equivalent implementation (executable code); and "validation" - correctness checking (at least, consistency and completeness necessary to generation) of system descriptions. Unifying these functions is an underlying semantic representation of system descriptions.

The conceptual model for the environment's program family structures both the user interface for the basic functions and the underlying representation. The variations allowed by an abstraction are represented by the concrete data items associated with each concept. Generation is free to transform specifications into any suitable structure (the internal system design), possibly constrained by specifications of required operational characteristics.

2.2 Functional Specification Formalization

"Specifications" of an application system in an abstraction-based environment are semi-formal descriptions of the distinctive characteristics of the system relative to the characteristics of the associated family. Semi-formal descriptions rely on constrained textual and graphic presentations. Deficiencies identified by [Meyer] that recur in informal requirements specifications are easily avoided with a semi-formal specification. Further, semi-formal descriptions are sufficiently unambiguous to allow automatic derivation of programs, while by avoiding formal, mathematically rigorous descriptions, specifications are easier to construct and understand.

[Balzer] is a reasonable definition of the principles to which abstraction-based environments should adhere and the resulting implications for their design. Several fundamental principles are identified that are particularly relevant to abstraction-based environments: a system specification is a cognitive model, a specification tolerates incompleteness and incremental enhancement, and information within a specification are localized and loosely coupled for ease of change. Of particular importance to successful system development is customer involvement, which is encouraged by an evolutionary development process that allows iterative review and refinement of system descriptions.

Beyond these principles, a system specification in an abstraction-based environment must be sufficiently complete and formal to allow automatic derivation of an implementation. The utility of abstraction-based development is the degree to which a developer can take technology for granted, operating totally within the conceptual framework of the environment. Because there can be alternative implementations for given specifications, well-defined support for refining implicit implementation decisions may be needed but should be cleanly distinguished from functional elements of the specification.

2.3 Levels of Abstraction

A basic insight concerning abstraction-based environments is the inverse relationship between flexibility and domain specificity. For practical purposes, the more implicit knowledge an environment has about particular domains, the less flexible the environment will be in accommodating unanticipated domains. A way around this problem is to support "levels of abstraction" in the environment.

Essentially, this means that the basic conceptual model and abstractions of the environment are generic and supportive of any conceivable application. Beyond this, an "application-specific" capability presents a developer with a conceptual model and abstractions that are tailored to particular domains of interest. This level reduces the effort required to produce an operational system, within the bounds of the model, by allowing the developer to construct more inherently meaningful system descriptions. The inherent meaning comes from syntactic and semantic knowledge of the relevant application domains with which the environment has been extended by an application-specific environment builder and the mechanisms for transforming between the application-specific and generic concepts. Just as the generic environment can be made extensible, so too can an application-specific environment, resulting in a further, less flexible layer of application-specific environment for even more application-oriented development. The primary effort, given such a capability for layered environments, will be in conceiving of an appropriate conceptual model and abstractions for the application developer.

There are also (hidden) levels of abstraction below the generic environment level. The lowest level is a module-organized, procedural language level that provides generalizations of conventional execution environments, such as display devices, data storage, and procedural logic. At this level, the fundamental principles of software engineering (information hiding and abstract interfaces) are applied. Concerns for efficiency and host/target environment independence and portability are addressed here.

An intermediate level, discussed further in the next section, provides the fundamental capability for definition and instantiation of modular abstractions as valid procedural level constructs.

2.4 Abstractions and Metaprogramming

The conceptual model for an abstraction-based environment provides semantic structuring to a set of concepts presented asfunctional abstractions. The goal of this emphasis on abstractions is to make consideration of the future evolution of software and the resulting ease of change a primary focus of its initial and continuing development; the normal emphasis on immediate need leads to software that is resistent to change. Each abstraction corresponds to some (one or more) abstract module implementation for which a concrete implementation can be derived given a description of the distinguishing features of an accommodated instance of the abstraction. An abstract

implementation establishes the common characteristics of all possible concrete instances; specific parameterization distinguishes a particular instance.

The key concern in the use of abstractions, at higher levels as well as at this level, is that "proper" abstractions be supported, where "proper" means that the "right" set of concrete instances are subsumed by the abstraction. An abstraction which is too narrow will have an inflexible implementation that is seldom useful; an abstraction which is too broad will either have an implementation which is too inefficient for practical use or require overly complex parameterization.

The notation for describing abstract module implementations is referred to as the "metaprogramming language". The primitive elements of the notation are concrete fragments of procedural level constructs. This notation supports powerful forms of parameterization and composition of concrete fragments and layered abstract instantiations. In addition to primitive elements as parameters, aggregate elements (corresponding to lists, structures, and simple and recursive discriminated unions of primitive elements) and analogous operations allow definition of complexly parameterized and instantiated abstractions. Compile-time constraints on parameters detect invalid uses of an abstraction, to increase the reliability of instantiated code.

3 THE SPECTRUM ENVIRONMENT

The Spectrum environment is a Software AE project investigating methods for the automation of software development, including extensive reuse of software and domain-independent specification of application systems. Spectrum embodies an application-independent conceptual model of the software it can be used to produce (it does not provide any application-specific abstractions). As such, its capabilities are more restricted than a general programming environment but more flexible than an application-specific environment would generally be.

The goal in developing Spectrum has been to investigate possible methods for order-of-magnitude productivity improvements in the development of application software. The selected application conceptual model is oriented to programs which are heavily interactive, requiring persistent data storage, sophisticated knowledge representation paradigms, and integration of external software. The chosen approach was to identify the distinguishing features of potential applications so that, by specifying the appropriate details, a particular application could be constructed. This approach has been successful in achieving substantial and effective reuse of abstract programs which in turn resulted in very low effort for the development of reliable application software.

Following discussion of the Spectrum development process and the principles upon which its specification environment is based, the abstractions of the world model and external interface partitions of the Spectrum application conceptual model will be outlined.

3.1 Process and Principles

Spectrum subsumes the design, implementation, and integration phases of the conventional software lifecycle. The requirements analysis, system test, and management activities remain largely unaffected by the use of Spectrum. The system design activity is significantly altered in that its primary focus becomes that of transforming identified requirements into the conceptual model of applications accommodated by Spectrum.

The Spectrum specification interface is based on several principles. The approach to software requirements specification described in [Heninger] was a significant influence. System descriptions are partitioned into "world model" and "external interface" descriptions. The world model partition provides abstractions for representing the developers' concept of the "real world" for which the automated system is a model. These abstractions are sufficient for production of all data management and internal processing mechanisms of the resulting software. Based on a principle of output-directed processing, the external interface provides abstractions for modelling the processes and products by which the automated system affects the "real world". Abstractions for inputs are viewed as subordinate to internal, external interface, or external (real world) processes that trigger necessary input processing. Within each description partition, the principle of separation of concerns guides the design of particular specification paradigms appropriate to the various concepts.

3.2 World Model Abstractions

An application world model consists of a set of classes, packages, and strategies. Classes support modelling of static domain features. Concrete data associated with the objects of a class are characterized by data type; packages provide a medium for extending supplied primitive types with domain-specific abstract types (as well as for abstracting useful functional concepts). Strategies support modelling of dynamic features (logic and processes) of the domain.

A class is a representation of some category of real-world entities or information of value in understanding or describing the application domain. The model for representing a class in Spectrum is a semantic data model [Borgida], implemented to provide persistent storage of objects. Class definitions are organized into a (weak multiple) inheritance hierarchy that permits object membership in multiple sibling classes. Class members are characterized by a common set of features, referred to as attributes, that are represented by concrete data storage. An attribute is typed to contain either discrete data values or identifiers of related objects which are selected members of some specified class.

Packages provide a medium for developer definition of application-specific types and associated operations. Packages provide a more primitive alternative to classes for abstracting the characteristics and manipulation of data.

Strategies are based on concepts of knowledge-based and object-oriented programming ([AI], [Stefik]). A class may have associated strategies that define (1) how to construct the membership of the class (e.g., from an external relational database), (2) which attributes must be initialized upon object creation, and (3) under what conditions an object should be specialized from or generalized to a parent class.

An attribute may have associated strategies that define how the value of the attribute is determined. Strategies are comprised of one or several knowledge sources, combined to return either the first or all results from constituent sources. A knowledge source for a value attribute can be a procedure, a production rule set, the end-user, or an external device input. A knowledge source for a relation attribute can be a predicate-based selection from candidate objects, a request for a uniquely generated object, a user selection from candidate objects, or a selection from candidate objects by external device input. Any attribute can have as a knowledge source the strategy inherited for the attribute from an ancestor class or a nested compound strategy. Each knowledge source has an associated precondition that constrains its applicability and a postevaluation that constrains its

result. Attribute strategies can have associated resolution directions for discriminating among multiple results. Both class and attribute strategies are supported by a data model dependency maintenance mechanism that maintains consistency by requiring re-execution whenever (class-, object-, or attribute-level) source information changes.

Autonomous strategies provide a mechanism for describing autonomous data processes that can operate over the entire data model. These strategies provide for a process-oriented view of mechanisms that operate independently of data flows that are reported to the data model. All strategies, including class and attribute strategies, can be activated either by direct invocation, on system startup, periodically, or in response to events resulting from actions in the world model or from processing of asynchronous inputs from the external world.

3.3 External Interface Abstractions

An application external interface consists of a set of logical devices and a set of output managers. Devices characterize the logical media that connect the application software with the real-world. Output managers either transform world model data into outputs and out of inputs or control and coordinate other managers toward this end.

Devices are categorized, by media conceptual type, as standard hardware (CRT), external software (standard procedural or relational database), or (in the future) non-standard hardware. Devices have associated input and output data item definitions, and for relational databases a (partial) schema definition.

All output managers support selection of a world model object set context, management of associated enabled inputs, and optional event-based activations. Output managers provide for either output generation or output control.

An output generator for standard hardware conforms its context to fit a presentation paradigm, which can be object-oriented, relationship-oriented, text/document, or iconic/graphic; each presentation can contain nested presentations of any of these paradigms. All concrete data must be simple textual or iconic in presentation. Presentations are designed to emphasize abstract structure over format and direct manipulation of displayed representatives of data objects. Output generators have access to the world model dependency maintenance mechanism resulting in automatic redisplay of presentations that contain modified data. For external software devices, an output generator maps context into either external program parameters (for standard software) or relational query slots (forrelational databases).

Output controllers support various forms of output manager aggregation: sequencing, selection, concurrent, and subordinate.

4 RESULTS

The benefits of an abstraction-based environments has been clearly demonstrated in the use of Spectrum: in the development of two medium-sized (prototype) applications, an order-of-magnitude productivity gain was actually achieved. The first application supported work process management and document production; the second supported mail message management with transparent data access to a remote relational database server. Of 15-20 major abstractions implemented (each requiring 6 to 18 months of development effort), the first application reused ten that had been implemented for the Spectrum specification environment and required the

development of two new ones; the second application reused eleven of these, requiring major enhancements to two and three new ones. Apart from additional work required on abstractions, each application took less than a year of effort from conception to operation. For each, functional changes required insignificant effort and resulting code was unusually reliable, due to reuse, with the predominance of detected faults due to specification errors.

Despite this success, attributable totally to the emphasis on abstraction and attendant software reuse, constructing an environment adequate to the description of arbitrarily selected applications would require significant further effort. Since the primary effort is in the development of suitable abstractions, the most likely short-term benefit will come from constructing environments whose abstractions are tailored to the conceptual needs of particular application domains, with a bias to abstractions that support multiple domains. [Levy] established the viability of constructing application generators as an integral part of application development. Abstraction-based environments provide the basis for generalizing that approach to application program families.

5 RELATED AND FUTURE WORK

Abstraction-based environments provide a powerful approach to the construction of software. The emphasis on an integrating conceptual model and constituent abstractions is supported by an underlying metaprogramming capability that emphasizes flexibility and direct control of the instantiation process by the implementor of an abstraction. In principle, it encompasses many of the other approaches to improving software productivity.

[Boehm] surveys most of the recent approaches to improving software productivity, including fourth-generation languages and application generators, knowledge-based assistants and automatic programming, and component reuse. These approaches share many of the goals and benefits of abstraction-based environments. Current application generators ([Horowitz]) are generally restricted to narrowly defined domains, are based on high-level but relatively data-processing-oriented concepts, and offer high productivity. Knowledge-based approaches (e.g., [Smith]) tend to focus on automating the process of software development with only a secondary concern for the specifics of its abstractions and products; such an approach is beneficial but premature for production use. An object-oriented approach has been advocated ([Meyer2] and [Kaiser]) as a viable alternative that facilitates reuse. The underlying technology of past Spectrum work is most related to this approach but has a more conventional procedural language orientation. The specification environment work is similar to an application generator approach but exploits more sophisticated interactive technology and different code construction techniques.

Future work on abstraction-based environments should address more of the total software development process (management, system specification design, system test, and documentation). For maximum productivity gains, the entire process should be adapted to benefit from the use of abstraction.

Approaches to support dynamic validation and performance analysis of descriptive specifications are needed to make abstraction-based environments practical for production use. To be practical, abstract specifications must be the focus of all development related activities.

Alternative conceptual models should be investigated, particularly for accommodation of concurrent and embedded system concepts. The model designed for Spectrum seems very flexible but has not been applied to sufficiently diverse problems for a full evaluation.

An understanding is needed of the nature of domain-specific conceptual models and the viability of basing automation on transformation to a general, domain-independent model. It is not clear whether direct implementations of domain-specific abstractions are necessary or if they can be mapped practically into generic abstractions, which would be far more cost-effective.

6 REFERENCES

[AI] M. Stefik, J. Aikens, R. Balzer, J. Benoit, L. Birnbaum, F. Hayes-Roth, and E. Sacerdoti, "The Organization of Expert Systems", Artificial Intelligence, March 1982.

[Balzer] Robert Balzer and Neil Goldman, "Principles of Good Software Specification and their Implications for Specification Languages", USC/Information Sciences Institute.

[Boehm] Barry W. Boehm, "Improving Software Productivity", IEEE Computer, September 1987.

[Borgida] Alexander Borgida, "Features of Languages for the Development of Information Systems at the Conceptual Level", IEEE Software, January 1985.

[Heninger] Kathryn L. Heninger, "Specifying Software Requirements for Complex Systems: New Techniques and Their Application", IEEE Transactions on Software Engineering.

[Horowitz] Ellis Horowitz, Alfons Kemper, and Balaji Narasimhan, "A Survey of Applications Generators", IEEE Software, January 1985.

[IEEE] Special issue on Multiparadigm Languages and Environments, IEEE Software, January 1986.

[Kaiser] Gail E. Kaiser and David Garlan, "Melding Software Systems from Reusable Building Blocks", IEEE Software, July 1987.

[Levy] Leon Levy, "A Metaprogramming Method and Its Economic Justification", IEEE Transactions on Software Engineering, February 1986.

[Meyer] Bertrand Meyer, "On Formalism in Specifications", IEEE Software, January 1985.

[Meyer2] Bertrand Meyer, "Reusability: The Case for Object-Oriented Design", IEEE Software, March 1987.

[Parnas] David L. Parnas, "On the Design and Development of Program Families", IEEE Transactions on Software Engineering, March 1976.

[Smith] D. R. Smith, G. B. Kotik, and S. J. Westfold, "Research on Knowledge-Based Software at Kestrel Institute", IEEE Transactions on Software Engineering, November 1985.

[Stefik] Mark Stefik and Daniel G. Bobrow, "Object-Oriented Programming: Themes and Variations", The AI Magazine, Winter 1986.