Support for Change in RPDE^3

Harold Ossher and William Harrison
IBM T.J. Watson Research Center

Abstract: RPDE^3 is a framework for building environments. Great emphasis has been placed on supporting changes of various kinds, such as extensions to existing environments and creation of new environments by adapting existing environments. We have a three-pronged approach to supporting change: (1) use of a central framework providing key services is a uniform fashion, (2) an extended object-oriented programming paradigm supporting fine-grained changes by addition of small code fragments, and (3) structured representation of program material facilitating sophisticated language-sensitive processing. RPDE^3 has been used on a daily basis for its own development for about three years now, and during that time has undergone extensive change. This experience has indicated strongly that our approach to supporting change is effective, and has identified extensions to it that should make it more effective still. This paper describes the approach and, primarily, our experience.

Introduction

Successful environments are long-lived, being used by a number of people over an extended period of time for the development and maintenance of a variety of software systems. Developers' tastes and styles vary. Differences in systems impose different requirements on the environment. New techniques, new tools, new needs, and new hardware must be supported. Accordingly, adaptability and extensibility are essential features of a modern environment.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1990 ACM 0-89791-418-X/90/0012-0218...$1.50

RPDE^3 is a framework for building integrated, direct-manipulation environments. Entities manipulated by the environments are represented as objects. The user sees a display of objects on the screen, and can interact with them directly. When the cursor is positioned on an object, the user can issue any command appropriate for that object, irrespective of the "environment" or "tool" to which the command belongs. In fact, though we think and speak of multiple environments for convenience, all the environments are really part of a single, integrated environment. As the user moves to different objects, different commands become available without any explicit mode change or tool selection. This makes for seamless integration despite a possibly wide diversity of objects.

A major design goal of RPDE^3 was to support adaptation and extension of its environments while preserving the uniformity and integration just described.

For approximately three years now, RPDE^3 environments have been used on a daily basis by our group for the development and maintenance of RPDE^3 itself. During this period also, two other groups in IBM adapted one of our environments to their needs. A great deal of change and evolution have occurred. The primary kinds of change and a few specific examples are:

- *Extending function.* Providing greater functionality within an existing environment. For example, we added def-use resolution to an editing environment for programs, introduced hypertext-like links that can be used between arbitrary objects in existing environments, and introduced a version object that can maintain multiple versions of arbitrary objects.
- *Extending the data domain.* Extending the existing functionality provided by an environment to a wider selection of data (objects). For example,

we adapted a structural editing environment for Pascal programs to structural editing environments for other domains, including a specification language [3.], grid structure specifications [16.], Unix(TM) documentation and simple box-and-arrow diagrams.

- *Supporting new languages,* and a mix of languages, for software development. We adapted the environment for programming using PC Pascal to the VS Pascal dialect and to C. Other groups within IBM adapted it to two different dialects of PL/I. We now have an integrated programming environment supporting all these languages.
- *Integrating separate environments,* allowing RPDE^3 environments developed separately to be integrated after the fact.[1] We integrated our environments for programming in Pascal and C with those developed elsewhere for dialects of PL/I. Though we all started from the same base, there was considerable divergence during the course of development.
- *Supporting new hardware/operating systems,* allowing environments to be ported to new platforms. We used RPDE^3 to port itself from the IBM PC under DOS to the PC and PS/2 under OS/2 and the RT under AIX. Our environments now run on all three platforms, and data exchange among them is supported.

Many of these changes were straightforward, confirming our claim that RPDE^3 supports change effectively. Others were more difficult, leading to insights and improvements that should make similar changes easier in future.

This paper gives a brief overview of the RPDE^3 technology for supporting change, and then discusses our experience with a representative selection of the changes that have occurred during the life of RPDE^3. The manner in which the RPDE^3 architecture facilitated the changes is explained. Other conclusions drawn from this experience can be found in [6.] and [13.].

Technology

Support for change in RPDE^3 rests on three pillars:

- *Framework architecture.* The central RPDE^3 framework [7.] provides a collection of important services to all RPDE^3 environments. These include an object repository, command and keystroke management, display construction, information propagation around object networks, object-based mark/move/copy and an undo facility. This reduces work in building or extending an environment, is key to ensuring uniformity and integration, and hides the details of hardware and operating system from individual environments.
- *Extended object-oriented technology.* An environment is built by defining object types appropriate to the domain of the environment. These types must supply methods required by the RPDE^3 framework, and they make use of the services provided by the framework. We use the standard object-oriented polymorphic method call, and reuse through inheritance. However, we have extended the object-oriented paradigm in a number of ways to allow greater reuse and inheritance at finer levels of granularity than is usually available. This is key to facilitating adaptability and extensibility, allowing most changes to be made by adding new code in separate, small fragments rather than by modifying existing code.
- *Structured representation of programs.* All program material, whether the system being developed using an RPDE^3 environment or the code and definitions making up the RPDE^3 framework and object types themselves, is represented as structured objects rather than text. This greatly facilitates sophisticated language-sensitive processing, which is key to supporting new and mixed languages and language-related function.

Examples of how these aspects of the RPDE^3 architecture support change will be given in Section 3. Details of semantics and implementation are beyond the scope of this paper, but are given in a number of other publications [6., 7., 8., 9., 10., 11., 12., 17.].

[1] Foreign tools, those not using the RPDE^3 persistent object store, are coupled into the RPDE^3 environment by writing import or export filters. The existing prototype has commands for generating compilable source text, or parsing existing Pascal programs.

Experience

This section describes our experience with a representative selection of instances of the various kinds of change listed in Section 1.

Adding new function

Def-Use Resolution

The oldest RPDE^3 environment is one for Pascal programming using refinement. All Pascal constructs, such as procedures, declarations and statements, are represented as objects. The core of this environment permits editing of these objects and generation of textual Pascal in the PC Pascal dialect for submission to a compiler. A number of extensions have been built upon this core including def-use resolution, which will be further extended to full static-semantic checking.

Def-use resolution was implemented by providing commands "definition" and "nextuse" for finding the definition and next use of a symbol. Both commands respect the scoping structure of Pascal. These commands were built by associating *handlers* for them with object types that can contain symbols, such as expressions and statements. The handlers are methods, implemented using *structure-bound messages* [10.] that navigate the object representation of the program to find the site of the definition or next use. A key feature of structure-bound messages is that they facilitate communication between far-flung objects without the involvement of those objects that happen to be between them; default routing specifications control how messages navigate past uninterested objects. Thus only objects defining scopes (such as procedures and modules) and objects representing declarations and symbol tables needed to handle the def-use messages. In all, def-use resolution was implemented by adding 6 command handlers, 4 message types, and 10 message handlers. Note that addition of code fragments sufficed; no code had to be modified.

Hypertext

Given a number of existing environments, such as the environment for Pascal programming, we desired the ability to establish links among objects and to navigate those links. We wished to do this in a general way without having to make detailed modifications to existing objects or object types and without disrupting existing or future environment functionality, yet with the ability to link arbitrary objects.

We accomplished this extension by introducing some new object types. "Envelope" objects can be placed around arbitrary objects while containing additional data. Linking is done by means of new "link" objects that are placed in the additional data fields of envelopes surrounding the objects to be linked.

Envelopes are objects that are almost always transparent. They achieve this transparency by passing on method calls unchanged to the objects they surround. However, they have the freedom to trap any desired calls, and process them before, after or instead of the surrounded object. For example, the envelopes containing hypertext links handle calls to do with linking, usually by passing them to the link objects, but pass all other calls through to the surrounded object untouched. All objects used in our environments maintain explicit pointers to their parents and components, making it possible to surround any object with an envelope. Envelopes thus provide a way of extending the functionality and even the state of an object without modifying the representation or code of the object itself. They can be, and have been, used in many ways in addition to hypertext, such as to attach annotations and compilation error messages to objects and to record the files to which textual program code is to be generated.

The hypertext extension, including envelopes, was implemented using 10 new object types. Of these 10, only 2 were novel enough to require building display or other housekeeping methods, of which 14 were built. The remaining operations were built using inherited methods, tailored to define initialization or display behavior with 16 option packages. The hypertext extensions also called for 5 new command handling methods and 5 new message handling methods for the 3 new message types introduced. In addition, the introduction of the hypertext extensions created 8 new operations for which 23 methods were provided in connection with other pre-existing object types to simplify future extension.

Extending the data domain

Structural Editing Environment for Grid Specifications

A major part of the RPDE[3] Pascal environment is support for the basic structural editing functions: reading and saving files, display, structural navigation, text entry, text locate, and so on. These functions are available for the set of object types making up the Pascal environment. Building structural editing environments for other domains can therefore be seen as extending this function to new kinds of data. This section discusses the case of grid structure specifications.

The *grid* [16.] is a mechanism for specifying the structure of large, layered systems. It organizes program *units* in a two-dimensional matrix. The dimensions usually represent system component and layer The system components and layers (matrix rows and columns) are organized in hierarchies called *directories*. *Specifiers* and *qualifiers* in the directories are used to specify formally permitted interactions among units. Grid specifications are best rendered graphically. We built a structural editor for grid specifications that represents the various components of a specification (matrix, directories, specifiers, etc.) as objects, displays them graphically and permits the user to edit them.

Most of the structural editing operations are implemented centrally in the framework or in system commands (such as the "locate" command). The implementations call methods for object-specific processing. The methods are generally much simpler than the central code. The advantage of this standard object-oriented approach is that the complex functions are automatically extended to any new object type that supports the simpler methods.

Support for methods is usually provided, at least to some extent, through reuse. We have an ever-increasing library of object types, most general purpose, some specific to particular environments. There are a number of ways in which this library can be exploited when implementing a new type. They are listed here in increasing order of the amount of work required:

- *Inheritance*. In the vast majority of cases, a method can simply be inherited from the supertype. One of our object-oriented extensions, called *options* [8.], greatly increases the frequency with which this works. A pack of options is a collection of constants defined and referenced by a method, but associated with an object type. They tailor the behavior of the method. It is often possible to inherit the code and tailor it to new requirements merely by changing the options. Such tailoring is much easier and more intuitive than writing code, and lends itself to a direct manipulation, visual interface. In the case of the grid environment, the directories were built entirely from subclasses of existing box-like, list, text and link objects, tailored using options. No new method code was written for the directories.
- *Direct reuse*. In some cases, the method associated with the supertype is not suitable, but another method in the system is. That method can be named and used. As in the case of inheritance, tailoring can be performed using options. The applicability of this approach is increased by the association of instance variable declarations with methods rather than type definitions, lessening the dependence between the method code and the object representation [11.].
- *Components*. Most objects contain components. It is often possible to implement an operation simply by passing it on to the appropriate component. This leads to a design approach in which an object is given functionality by giving it suitable components rather than by implementing the functionality directly for the object itself. For example, specifiers in the grid environment are basically annotated links between directory nodes. We implemented them using link objects from the hypertext environment as components, rather than providing the specifiers with link functionality directly.

When an object receives a structure-bound message, it can route it to any or all of its components easily. When a command is issued or a method is called, however, routing to a component must be done by explicit code. Our experience has identified the need for simple routing mechanisms in these cases also. They should not be difficult to implement.

- *Rework* (*white-box reuse*). Even if a method implementation does not currently exist that can be tailored through options to produce the desired result, it is often possible to find a similar implementation and rework it. The result might be a new, special purpose implementation. Often, however, the result is a more general implementation that can now be tailored to both the old uses and the new one through options.

The portion of the grid environment that did not closely resemble any existing objects was the matrix. The approach of "building with components" was used heavily. Since most grid matrices are very sparse, a linked representation was used. This enabled dimensions to be represented as lists of slices, and slices as lists of entries, using variants of existing list objects. Much of the behavior of these lists was inherited, but some of it, especially display-related behavior, had to be coded. The approach to writing the methods was always to rework the method that would have been inherited. The extent of the changes varied considerably, but in all cases some of the original code was left, and using it as a starting point was a significant help. The matrix and matrix entry types themselves were built from scratch. Even here, however, existing method implementations were used as the starting point for each new one.

Much of our current and future research is directed at the problems of finding appropriate methods to rework [15.,17.] and at performing the rework in a way that leads to reusable abstractions and that keeps track of the relationships between the fragments of code involved so as to cope elegantly with subsequent changes.

In all, the grid environment was implemented using 41 new object types. Of these 41, only 4 were novel enough to require building display or other housekeeping methods, of which 36 were built. Three other types required the building or reuse of a single method in addition to inherited methods. The remaining operations were built using inherited methods, tailored to define initialization or display behavior with option packages. The hypertext extensions also called for 8 new command handling methods for the 100 new commands introduced, and 5 new message handling methods for 5 new message types introduced.

An important issue in extending the data domain is whether a new type of object will fit gracefully in existing contexts. For example, the envelopes supporting hypertext and the version objects mentioned earlier are intended for use in existing environments. If one of these objects is inserted by the user in the midst of, say, a program, it must not cause the existing environment for manipulating the program to break. We support the ability to use new objects in existing contexts, or to use objects in unanticipated contexts, primarily through:

- Structure-bound messages, which successfully pass by objects that know nothing about them.
- Use of *roles* [12.]. A role is a collection of operation interfaces, and is analogous to an abstract type in Emerald [1.]. When specifying what kind of object is needed in a particular context, such as in a particular instance variable or parameter, we specify a role rather than a specific type. Any type that supports the role, by providing implementations for the operations in the role, is acceptable. The types supporting a role can be widely scattered across the inheritance hierarchy, so roles are more flexible than abstract or virtual classes [4., 21.]
- Use of standard roles. Many functions are implemented only in terms of the operations in a few standard roles that are supported by all object types in our system. Such functions are therefore universally applicable.
- Avoiding code that queries the type of an object directly. Type names cannot be used as literals in the code so that objects do not become dependent on the types of their components or containers. Code can query role support or *properties* of objects, but these can be supported by multiple types.

It should be clear from the examples above that adding new function and adding new data often go together. Adding hypertext functionality required introducing a number of new object types. The grid environment as built and described added no new function, but the next step in building a full grid environment would be to add interaction checking. Most enhancements thus involve addition of some new object types and some new functionality.

Supporting new variations

The "nopackaging" perspective

Program material in RPDE[3] environments is represented as objects in a database rather than as collections of files. When source code is generated from the object representation for processing by foreign tools, however, it must be stored in files. We surround objects whose source code must go in a separate file by an envelope called a "sourcepackage". This envelope contains the file name and controls the source-code generation process.

Normally, sourcepackage envelopes are visible to the user when browsing a program. They can create considerable clutter, however, that is of no interest when one is dealing with program logic. Accordingly, we provide a display *perspective* in which the sourcepackages are not displayed. The user specifies the perspective desired by setting an environment variable. The alternative perspectives are implemented by using another of our object-oriented extensions, called *subdivisions* [9.]. Our support for method call can select the appropriate method to execute at call time based not only on object type but also on another, programmer-specified criterion. In the case of display-related methods, this criterion is the value of the perspective variable. Most objects display identically in the default and nopackaging perspectives, so the nopackaging subdivision simply inherits the default implementations. Only the two variants of sourcepackage objects display differently. Even these do not require override methods; they inherit the code but set two options differently in the two subdivisions.

Alternative perspectives have been used in like manner in other contexts, such as "browse" and "button" perspectives for user-interface buttons. Browse perspective hides the script associated with buttons, whereas button perspective display it. The difference, once again, is controlled by two simple options. Since just display perspective is involved, the functionality of the buttons is not changed at all. This allows the user to change and test the button script without repeatedly changing modes.

Further details of perspectives and their implementation by means of subdivisions are given in [9., 17.].

Supporting new languages

We adapted the Pascal programming environment to environments for other procedural languages, and then integrated these into a single mixed-language environment. This section describes the evolution of these environments.

Dialects of Pascal

We ported RPDE^3 from the IBM PC supporting PC Pascal to the RT supporting VS Pascal. This port involved many of the usual porting activities, such as rewriting low-level layers of the framework. We employed *version objects* at both the module and the detailed code level to maintain these different versions within a single, integrated, object representation of the system. The most interesting aspect of the port, however, was coping with the substantially different dialects of Pascal. In this context, it was not sufficient to provide a separate environment for writing VS Pascal programs. We needed to be able to generate VS Pascal source code from our existing object representation of some 30,000 lines of PC Pascal code, and then to maintain both versions in parallel.

Our approach was to introduce the capability to generate both PC and VS Pascal source from the same objects. At a high level, this was easy to do in RPDE^3. The generation of source code is done by "code-generate" methods. An initial set, highly option-driven, generated PC Pascal code. By introducing an environment variable called "target-language" and subdividing the code-generate operation on its value, we created the ability to generate different source code for different target languages. It is worth noting that subdividing an operation in this fashion in no way invalidates existing methods. The original methods become the default ones, and apply to all subdivisions unless overridden. For those constructs that look different in VS Pascal, subdivision overrides were therefore needed. There are 60 types of objects for which code can be generated in Pascal. For 11 of these types of objects, the method for code generation was used unchanged and unparameterized. For the remaining 49 types of objects, only 4 new methods were written, 12 existing methods were employed using different options, and (because of the similarities in the two Pascals) 33 methods were employed with identical options.

Generating the different dialects for structured constructs like procedures, conditionals and loops was especially easy. Not so the handling of assignment statements and expressions. For example, different names are used for the same builtins in the two dialects, and the maximum length of symbol names differs. A simple symbol parser and translator, invoked by the code-generate method of statement and expression objects, served to deal with this. Greater difficulties were caused by the fact that VS Pascal is a weaker dialect than PC Pascal in some ways. For example, it does not have "short-circuit" conditional expressions, and it does not permit dereferencing of the return values of functions. We provide these features in our environment by performing transformations involving temporary variables and additional statements. A few features, such as the PC Pascal "retype" function for type coercion, proved to be more difficult to support than we thought worthwhile. We discourage use of these features in our environment, requiring the user to employ alternatives if the code is to be portable.

The features of RPDE^3 that contributed most to the implementation of the integrated PC/VS Pascal environment were the structured representation of program material and the subdivision and option mechanisms. These can also be used to support language extensions. New objects can be defined to represent new language constructs. Their graphical syntax is specified by their display methods and options. Their semantics is specified by code-generation methods and options.

C

We also adapted the Pascal programming environment to an environment for C programming. Since the structure of Pascal and C is quite similar, our approach was to keep substantially the same object types but to generate C source code from them. Unlike the case of the two Pascal dialects, we do not attempt to transform assignment statements and expressions. The user of the C environment enters these in C, and they are generated unchanged. Some detailed changes were made to type declarations to take account of the different data types supported by C. These changes were made with options.

The semantics of some constructs are different in C from Pascal. For example, the C "switch" allows control to fall through to the cases following the selected one, whereas the Pascal "case" allows exactly one case to be executed. Using the same construct for such different semantics would be confusing to the user. Accordingly, we introduced a new object type for "switch", available in the C environment. We also allow the Pascal-style "case" to be used in the C environment, generating C source code that correctly realizes the Pascal semantics. There are also some additional constructs available in C but not in Pascal, such as the C looping construct. They are represented by 11 new object types in all.

The Pascal and C environments thus share many object types, a great deal of code and many options, but there are many differences in options and some differences in object types. The subdivision mechanism is used to select which alternative to use in each case. In addition to the "target-language" environment variable mentioned above, there is also a "source-language" environment variable that the user can set. Object initialization and much option retrieval are subdivided on the contents of source-language. Certain commands are specific to source-language also, such as the command to convert a statement to a switch. The RPDE^3 command handler supports *command filtering*: the ability to restrict the availability of a command based on object-specific criteria, such as source-language.

In fact, there are 64 types of objects for which code can be generated in C. For 11 of these types of objects, the method for code generation was used unchanged and unparameterized. For the remaining 53 types of objects, only 2 new methods were written. They, along with a few of the earlier code-generation methods, were reused 51 times with different options producing different behavior.

Dialects of PL/I

While the RPDE^3 group was enhancing the framework and various environments, two other groups successively built environments for two dialects of PL/1, based upon an isolated copy of our Pascal environment. About a year after exporting the copy, we took the second of these PL/1 environments and reintegrated it into our system. Subdivisions were a major factor in the reintegration effort, allowing the PL/1 additions to be kept separate from our code, and yet be incorporated into our system. The separation of framework from object types also stood us in good stead. We had made substantial enhancements to the framework, including porting it from one platform to another as described earlier. Once the PL/1 changes were integrated, the environment ran successfully under the new framework.

In fact, there are 70 types of objects for which code can be generated in the PL/1 dialect. For 11 of these types of objects, the method for code generation was used unchanged and unparameterized. For the remaining 59 types of objects, only 15 old methods were renamed and reintegrated. The remaining 44 objects employed existing methods, using options to provide the appropriate behavior.

Our main lesson from this experience was that it is not wise to allow in-place modification of copies of the system, because they become difficult to distinguish and reintegrate afterwards. Rather, if a person or group wishes to implement environment x, it is better to provide a copy of the current system with x subdivisions defined, and with a distinguished name space for x types, options and methods. Implementation of x then proceeds entirely by adding object types and by adding options and code in the x subdivisions. Because of the distinguished name space, most reintegration becomes trivial.

There remain three aspects of reintegration that are not trivial:

- Changes to the framework. Fortunately, most environment designers do not make such changes, and as the framework becomes more advanced and stable, they are needed less and less. However, cooperation with the controlling group is definitely desirable in the case of such changes.
- Changes to existing types. When this involves mere addition of methods, it presents no problem. When it involves modification, however, the result is difficult to integrate because of possible clashes with parallel modifications made elsewhere. When it involves modification of the representation, existing instances in repositories become invalid and must be upgraded. Passing instance variables to methods indirectly is one way of alleviating this problem, as is described in [11.].
- Inappropriate creation of new types. If new types are created where existing ones, perhaps extended, would suffice, it becomes difficult to combine these types later. By that time, instances of both the old and new types exist, so both representation upgrading and detailed code integration are required. Creating new types is thus not an effective solution to dealing with the difficulty of changing existing types if recombination is anticipated. The new and old variants can coexist, but result in similar material being represented in more than one way.

Coordinating and integrating changes made by different groups is important in the successful reintegration of divergent software, as is described in [19.].

Common Declarations

In mixed-language programs, it is often necessary for procedures written in different languages to manipulate the same data. This is best accomplished by allowing them to import the same data declarations. However, data declarations differ substantially in different languages. In our programming environments, this was manifest as substantial differences in declaration options and commands. Users of the different environments were effectively working with different declarations even at the object level, and each could generate source code in only one target language.

We unified the declarations across all the languages by producing a restricted *standard subset* of declarations that can generate successfully all the target languages supported. Language-specific variants remain available. Different language-specific declarations for the same data item can be included within version objects discriminated on "target-language".

Unifying the declarations involved extensive examination and manipulation of options. Supporting generation of all the target languages from each declaration involved supplying different code-generation options in the subdivision for each language. The language-specific variants are available through commands that are filtered according to "source-language".

Analysis of about half the RPDE^3 system revealed that only 3% of existing declarations were not in the standard subset. Half of these were strings, which are treated differently in Pascal and C, and the others were Pascal subranges, which are not supported in C but can usually be rendered as the base types. This result encourages us to believe that the standard subset has real utility.

The possibility of generating include files for multiple languages from the same symbol table poses a configuration-management problem: how to name and where to put them. We devised *configuration description* objects to be associated with source packages, by means of hypertext links. They enumerate all the target languages that can be generated from the contents of the package, and for each one give details such as path, file name prefix and extension, and version.

Related Work

Related work falls primarily into two categories: environment generators and object-oriented application frameworks and toolkits.

Environment generators, such as the Cornell Synthesizer [20.] and Gandalf [5.], follow the approach of processing an environment definition, usually some form of attribute grammar to generate an environment. Since the same generator is used for all environments, a framework/environment split related to ours is achieved. Program material is usually represented in structured form as abstract syntax trees. It is usually presented to the user textually rather than graphically, but structured operations are supported.

The main difference between the environment generators and RPDE^3 is in the manner of making extensions and adaptations. In the case of environment generators, this is done by modifying the environment definition and regenerating the environment. Since this is essentially a declarative approach, it seems appealing, but it does have drawbacks. Environment definitions are long and complex and often intertwined, especially when the attributes of attribute grammars are considered. For example, adding a new attribute for performing some additional checking can require changes to attribute equations across much of the environment definition. Some kinds of changes tend to be localized and easy, such as adding a new alternative to a syntactic construct or changing the unparsing scheme to support a new dialect. Others, however, present major problems. For example, the addition of constructs like hypertext envelopes or version objects that can be used anywhere present major problems. Integrating extensions made independently is also no simple matter. Any change clearly requires access to the source of the environment definition, so that the changes can be made to it and the environment regenerated.

The main advantage we claim over the environment generators is that our use of subdivisions, options, structure-bound messages and roles make extension and adaptation of environments easier. The changes can be made by adding fragments of code, often just a few small fragments. This does not interfere with existing functionality, and does not even require access to the source code of existing fragments. Extensions are usually independent or nearly so, so that integration becomes easier.

Object-oriented application frameworks, such as the Smalltalk Model-View-Controller [4.], and toolkits such as Interviews [14.] and Andrew [2.] follow the object-oriented approach of growing systems by subclassing. They therefore have many of the advantages we claim for RPDE^3 with respect to extensibility. There are some important differences, however. The separation between framework and domain objects is usually not clear-cut. The framework is usually specified as high-level classes that are inherited by domain classes. This means that, though it does not have to be separately written, framework functionality ends up being mixed with domain functionality. Inherited methods can be overridden. While this allows greater flexibility, it means that the framework no longer guarantees uniformity and that changes to the framework can have a serious effect on existing applications. At a minimum, recompilation is required. Worse, user overrides might no longer work, and existing objects might have incompatible representations.

Unlike the generated environments, the application frameworks and toolkits do not usually employ a structured representation of programs below the module level. The class hierarchy tends to be treated structurally through a hierarchy browser, but code is treated as text. This makes language-sensitive processing more difficult to accomplish. Finally, our experience has demonstrated that our extensions to the object-oriented paradigm are important, permitting greater reuse and easier tailoring than is available in conventional object-oriented systems.

The research issues being pursued in connection with Garden [18.] are similar to those of interest to us in building and using RPDE^3. Both systems employ a substantial functional framework of services, with an object-orient definition of conceptual language structures on top. The developers of Garden have employed it to explore graphical programming, the creation of user concept structures, and visual output. We have employed RPDE^3 to explore ways of exploiting the structure of professionally developed software to solve in-the-large programming problems. We have also devised and demonstrated the usefulness of some extensions to the object-oriented paradigm. RPDE^3 is also successfully used for its own development on an ongoing basis.

Summary and Conclusions

We have described the RPDE^3 approach to supporting change, and our experience with many different kinds of change as RPDE^3 has evolved.

Our key contribution is the three-pronged approach to environment construction:

- The framework architecture. This permits uniformity and integration despite diversity of domain, and allows changes to be made to central services without affecting domain-related code. Throughout the framework, the introduction of object-type dependent controls like command filtering and the display construction protocols were used to permit flexible use of a consistent point-of-view.
- Object-oriented extensions supporting fine-grained reuse, extension by addition, and integration of extensions:
    - Subdivisions, permitting a single object type to have separate implementations of a single operation for different situations. New cases can be added without affecting existing cases. Defaults are inherited.
    - Options, permitting inherited methods to be tailored in a simple, declarative manner.
    - Structure-bound messages, permitting communication between far-flung objects without the involvement of the objects that happen to be between them.
    - Using roles and avoiding the mention of specific types in code. This permits the substitution of one object for another in any context provided both support the operations required by that context. Objects can often be used successfully in unanticipated contexts.
    - Decoupling of instance variable declarations from type definitions, permitting wider reuse of method code unconstrained by the inheritance hierarchy.
- Structured representation of programs. This facilitates sophisticated language-sensitive processing and the ability to handle multiple languages in an integrated fashion.

Our experience has shown that this approach is effective in supporting the many kinds of change that confront a modern environment, and has identified improvements and extensions that should make it more

References

1. Black A., Hutchinson N., Jul E., Levy H., Carter L., Distribution and Abstract Types in Emerald, IEEE Transactions on Software Engineering, Vol. SE13 No. 1, January 1987, pp. 65-76.
2. Borenstein, Nathaniel S., Multimedia Applications Development with the Andrew Toolkit, Pentice Hall, 1990. ISBN 0-13-036633-1.
3. Fern K., "Mondrian: A Two-Dimensional Graphical Specification and Design Programming Environment", Master's thesis, Massachusetts Institute of Technology, May 1988.
4. Goldberg A., Robson D., Smalltalk-80, The Language and Its Implementation, Addison Wesley, Reading, Ma., 1983.
5. Habermann A. N., Notkin D., Gandalf: Software Development Environments, IEEE Transactions on Software Engineering, December 1986, pp. 1117-1127.
6. Harrison W., Building Extendible Tools and Applications From Small Fragments, IBM Research Report RC 14533, March 1989.
7. Harrison W.H., The RPDE^3 Environment - A Framework for Integrating Tool Fragments, IEEE Software, November 1987.
8. Harrison W., RPDE^3 - An Environment Framework for Integrating Tool Fragments, IBM Research Report RC 12646, April 1987.
9. Harrison W. and Ossher H., Subdivided Procedures: A Language Extension Supporting Extensible Programming, Proceedings of the IEEE Computer Society 1990 International Conference on Computer Languages, March 1990.
10. Harrison W. and Ossher H., Structure-bound Messages, IBM Research Report RC 15539, March, 1990.
11. Harrison W. and Ossher H., Attaching Instance Variables to Method Realizations Instead of Classes, IBM Research Report RC 15538, March, 1990.
12. Harrison W., Ossher H., Checking Evolving Interfaces in the Presence of Persistent Objects, IBM Research Report RC 15520, February, 1990.
13. Harrison W., Shilling J., and Sweeney P., Good News, Bad News: Experience Building a Software Development Environment Using the Object-Oriented Paradigm, Proceedings of the 1989 Conference on Object-Oriented Program-

ming: Systems, Languages, and Applications, October 1989.

14. Linton M., Vlissides J., Calder P., Composing User Interfaces with InterViews, IEEE Computer Magazine, February, 1989.

15. Maarek Y. and Smadja F., Full Text Indexing Based on Lexical Relations. An Application: Software Libraries, Proceedings of SIGIR '89, 12th International Conference on Research and Development in Information Retrieval, June 1989.

16. Ossher H., A Mechanism for Specifying the Structure of Large, Layered Systems, in Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pp. 219-252, MIT Press, 1987.

17. Ossher H., Multi-Dimensional Organization and Browsing of Object-Oriented Systems. Proceedings of the IEEE Computer Society 1990 International Conference on Computer Languages, March 1990.

18. Reiss S.,Working in the Garden Environment for Conceptual Programming, IEEE Software, November 1987.

19. Reps T., Horwitz S., Prins J., Support for Integrating Program Variants in an Environment for Programming in the Large, Proceedings of the International Workshop on Software Version and Configuration Control, Teubner Verlag, Stuttgart, January 1988.

20. Reps T., Teitelbaum T., The Synthesizer Generator, Proceedings of ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments, April 1984, pp. 42-48.

21. Stroustrup B., The C++ Programming Language, Addison Wesley, Reading, Ma., 1986.