Toward Support for Structural Evolution in Exploratory Software Development

Philip Johnson

COINS Technical Report 89–110
February 1989

Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

This work is supported by the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, New York 13441-5700, the Air Force Office of Scientific Research, Bolling Air Force Base, District of Columbia 20332, under contract F30602-C-0008, supporting the Northeast Artificial Intelligence Consortium (NAIC).

Contents

1 Introduction

1.1 Structural Evolution in Exploratory Development

Structural evolution is common in software, but pervades exploratory software due to its *experimental* and *domain-embedded* nature[1]. Experimental software is a vehicle for discovering new computable solutions to problems. In contrast, non-experimental software is simply the implementation of a problem solution that was discovered before implementation began. Domain-embedded software is software that, once implemented, changes the nature of the problem to be solved. Experimental, domain-embedded software exhibits frequent and unpredictable evolution in its behavioral requirements.

Evolution in behavioral requirements induces structural evolution by establishing new structural goals. For example, software is normally more easily understood if its structure corresponds to its behavior. A common motivation for reorganization of exploratory software is to maintain this *structural-behavioral correspondence.*

New behavioral requirements for the exploratory system may be shared to some extent by another system, and the two projects may find it advantageous to develop or reuse a common subsystem. This goal of *exploitation of community resources* normally leads to structural evolution as well.

Yet another way in which behavioral evolution induces structural evolution is through the desire for *developmental asynchrony.* Developmental asynchrony occurs when the structure of the system allows each developer to work autonomously on the current set of behavioral enhancements without "stepping on one another's toes."

Unfortunately, these goals for software structure are often mutually exclusive. Shared subsystems normally involve compromises on both sides, leading to structural idiosyncrasies which are tolerated because of the benefits of sharing. Similarly, the structural organization best suited to asynchronous implementation of the current behavioral enhancements is rarely identical to the one which maximizes the structural-behavioral correspondence.

Ideally, the structure of an exploratory system would be fluid, adapting to the changing behavioral requirements and the changing goals for the structure. For example, when new developers join a project, the structure might evolve into close structural-behavioral correspondence to ease their transition. As they become familiar with the system, the structure might evolve away from this correspondence and toward one more supportive of asynchronous development. In reality, however, the structure of an exploratory system is difficult to change, and evolutionary behavior

[1]These terms are adapted from [Giddings, 1984].

of this sort rarely if ever occurs.

I believe that there are several important reasons why the structure of exploratory software fails to fluidly evolve in response to the changing goals of the development process.

First, there is relatively little support for the process of structural change—virtually any kind of structural reorganization must be accomplished by hand. Second, support for structural change requires representing software structure, and exploratory languages require different structural representations than extant representations, which were developed for more traditional languages. Third, changing the software structure, even to satisfy an important structural goal, means new information for developers to assimilate about the software. Highly evolutionary software structures introduce new overhead in the form of "keeping up" with the changing structure. In some cases at least, developers might opt for a familiar but idiosyncratic structure over an unfamiliar but "better" one.

This proposal describes the design of an environment to help address these problems of structural evolution in exploratory software. The environment provides automated support for common structural transformations in exploratory development, thus alleviating the overhead of manual change. It employs novel type and module representation mechanisms amenable to exploratory language features. It helps developers cope with increased structural fluidity by representing the history of structural evolution and provide techniques to help developers update their structural model of the system. Finally, I hope the environment will serve as a knowledge acquisition tool for processes of structural evolution. This knowledge can be used to develop new forms of automated assistance, as well as to learn about desirable and undesirable forms of structural evolution.

Although the structure of a system is intimately related to its purpose, my current focus for the environment is on domain-independent representations of knowledge about structure and processes of structural evolution. I believe that much useful knowledge about structural evolution can be captured in domain-independent form, and that this form has the advantage of immediate applicability to a wide variety of developmental contexts. In addition, it does not introduce overhead in specification and maintenance of semantic information about the purpose of systems and their components. Automated semantic analysis techniques (for example, [Letovsky, 1988]) can provide some of this information without incurring this overhead, however.

1.2 Organization of the proposal

To motivate my approach, the next section presents several examples of evolutionary phenomena in exploratory development, the structural problems they pose, and the forms of automated assistance I hope to provide. The following section discusses the organization of the environment—the organizing framework for these processes and structural representations. These two sections together lay out the core of my thesis: the specific genre of problems for which I hope to provide computable answers.

The next section contrasts my approach to other related work: structural evolution in software, software development methods, the software process, type systems, and domain-independent characterizations of evolutionary processes.

The penultimate section presents my research plan. This section outlines the contributions I expect this research to make, the amount of time I believe I'll need to make them, and the methods I intend to use to evaluate the contributions. The proposal concludes with a brief summary.

2 Problems and Processes of Structural Evolution

In the introduction, I argued that exploratory programming benefits from highly evolutionary software structure, but that this evolution is hampered by the lack of automated support. I also claimed that structural (as opposed to behavioral) knowledge can suffice to provide important forms of automated support for structural change. While compelling support for these hypotheses requires experience with these types of mechanisms, this section offers evidence for the plausibility of these hypotheses through four examples of structural support mechanisms based upon structural knowledge. Following these examples is a subsection listing some other promising forms of structural evolution for potential future research.

2.1 Object Emergence.

While some objects[2] are recognized and implemented as such in the initial implementation of a system, many others *emerge* from the convergent evolution of previously unrelated parts of the system over time.

For example, functions often implement "inline" data objects and operations that are used once and then thrown away when the function completes. Over time, if new contexts arise in which these sets of data objects and operations are applicable, structural aggregation as an object is preferable to simply copying the inline code to each of the new contexts. Objects provide obvious benefits over multiple occurrences

[2]I use "object" in the generic sense of an aggregation of functions and data supported through some structural construct such as `defstruct` or `defclass`

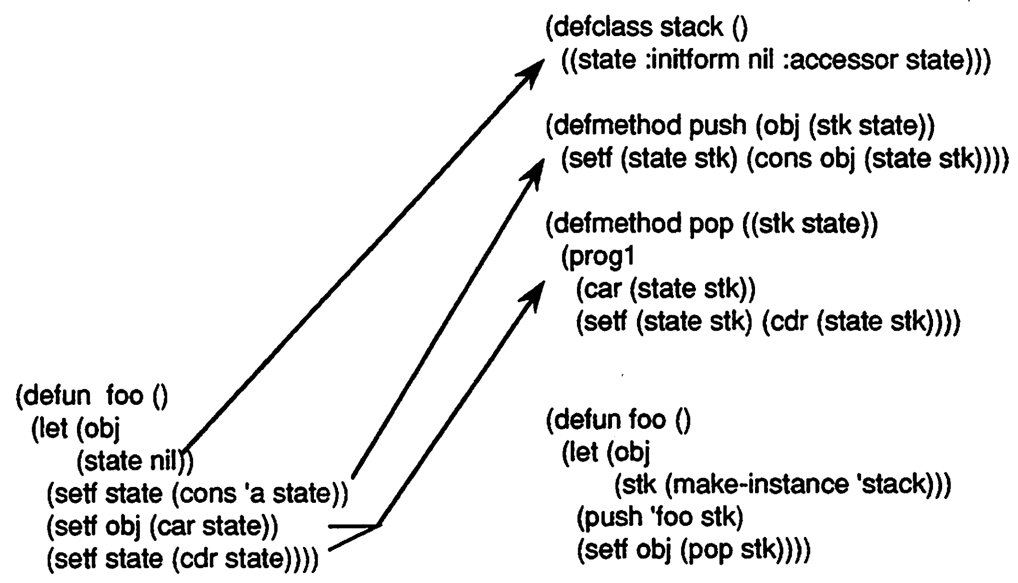
of inline code: the localization eliminates multiple updating; the common occurrences are explicitly recognized; and new instances can be created more easily.

Object emergence can also be induced by the introduction of object constructs into a language, which has occurred with the Common Lisp Object System, C++, and Object Pascal. For example, there are many Common Lisp systems which implement objects conceptually, but without any structural support from the language. A CLOS implementation would provide many developmental benefits by making these implicit objects explicit.

While object emergence is a beneficial developmental phenomena, the tasks involved are poorly supported or even unsupported in current environments. First, there are time-consuming but relatively straightforward "cutting and pasting" tasks: definition of the structural template; abstraction of inline code into functions and replacement of the code by function calls; creation and initialization of the structure instances; and replacement of inline variable references by calls to structure accessor functions. In addition, analysis is involved in inserting the new object into the existing object hierarchy: how should the hierarchy be changed in order to accommodate the new object?

My proposed environment can provide a great deal of support for the process of object emergence, simply by representing and exploiting structural knowledge about object language constructs and other forms of code, and by representing the sequence of tasks required to perform this type of structural evolution. First, the process has information about object constructs such as `structures` or `CLOS classes`, and how to construct their defining forms. Second, the environment supports a process based on lambda-abstraction for the creation of functions from inline code. Third, it encodes information about alternative ways in which an extant object hierarchy can integrate a new object. This information is integrated into a single **emerge-object** process.

The encoding of this knowledge changes the process of object emergence into a collaborative process between the environment and developer, rather than an unaided, manual undertaking. After the developer initiates the **emerge-object** process, the environment asks for the kind of structure to be created, labels for slots, sections of inline code to be transformed into methods, etc. It then creates the new object definition and replaces the old in-line code with the corresponding object references. Finally, the occurrence of this evolutionary process and the effect it had on the software structure is preserved in the environment as part of the evolutionary history of the system. These historical mechanisms are discussed further below. An example of the creation of a CLOS object from an inline function is illustrated in figure 1.



```
(defun foo ()
  (let (obj
        (state nil))
    (setf state (cons 'a state))
    (setf obj (car state))
    (setf state (cdr state))))
```

```
(defclass stack ()
  ((state :initform nil :accessor state)))

(defmethod push (obj (stk state))
  (setf (state stk) (cons obj (state stk))))

(defmethod pop ((stk state))
  (prog1
    (car (state stk))
    (setf (state stk) (cdr (state stk)))))


(defun foo ()
  (let (obj
        (stk (make-instance 'stack)))
    (push 'foo stk)
    (setf obj (pop stk))))
```

A Stylized, Before and After Example of Object Emergence.

The arrows point out how the inline code was extracted into the object definition. The original and restructured foo functions appear side by side.

Figure 1: An example of object emergence

2.2 Abstract Data Type Emergence.

Another general class of evolutionary processes relates to the emergence of abstract data types. While distinguishing between ADTs and objects goes beyond the scope of this paper, the two basic differences are: (1) the aggregated software entities in an ADT obey a *protocol* distinguishing an external public interface from an internal private representation; and (2) relationships between types are based upon shared semantics, while relationships between objects are based upon shared code. In exploratory development, an ADT often emerges from an object as experience with the object leads to the recognition of which components should be hidden (in order to ease their re-implementation, for example), and which define the public interface. ADTs emerge in other ways as well, such as from inline code, or from the union or fission of other ADTs.

While the **emerge-adt** process is similar to **emerge-object** in its general style of usage—the developer invokes it, the process prompts the developer for required information, restructures the code appropriately, and records the event—supporting ADTs in exploratory languages like Common Lisp poses several additional interesting problems.

While Common Lisp provides `defstruct` or `defclass` for object definition, it provides no constructs for explicit support of abstract data types. This may be a result of a more general problem: most language constructs for ADT specification are embedded within statically typed languages that, in addition, disallow dynamic function invocation. Although run-time type and function definition as well as mechanisms such as `funcall`, `eval`, and `apply` are prized advantages of exploratory languages like Common Lisp, they complicate enforcement of protocols for defining public and private interfaces.

Supporting abstract data type definition with the **emerge-adt** process thus involves representing knowledge about structure and process analogous to that used to provide automated support for object emergence. In addition, the environment must provide the structural constructs missing in Common Lisp for ADT definition and relationships, as well as mechanisms to enforce these protocols. Our environment provides a relatively simple protocol to distinguish the public and private components of an ADT. To enforce this protocol—-in other words, making sure private components stay private—I are designing a mechanism involving a mixture of static and dynamic mechanisms termed *demand-driven typing*.

2.3 Demand-driven typing

Demand-driven typing is a new approach to providing exploratory environments with the benefits of enforcable abstract interface mechanisms, without requiring everything to be typed, and without prohibiting dynamic function/type definition and invocation.

Very briefly, demand-driven typing for the purposes of ADT protocol enforcement proceeds as follows. First, it checks whether structural and type information obtainable through static analysis is sufficient to show conformance with the currently defined ADT protocols. It does this through constructing a representation of the control and data flow in the program which attempts to determine which type or set of types is passed to or returned by each function, and whether other types might be returned or passed whose identity cannot be determined through these static methods. Figure 2 illustrates an example of a function whose types can be determined through this process, and one whose types cannot be precisely identified.

Types can be determined

```
(defun foo (bar)
  (cond ((null (car bar))
         (cdr bar))
        ((null bar)
         "error")))
```

Types cannot be determined

```
(defun foo-2 (baz z)
  (funcall baz z))
```

Demand-based typing examples.
For foo, demand-based typing can infer the type of bar as a list, and the returned types of foo to be either a string or a list.
For foo-2, it can only determine that baz is a function that can be passed a single argument, and it can infer nothing about z.
If more information than this was necessary to enforce an ADT protocol, it would be requested from the user.

Figure 2: Examples of Demand Based Typing

If this static analysis of the program is not sufficient to demonstrate that the ADT protocol is enforced, then the missing type information is requested from the user. If this additional information is indeed sufficient to enforce the protocol, then run-time enforcement of the user supplied type information is compiled into the code, and the process ends there.

If the user is unable to supply enough type information to enforce the protocol, then as a last resort, the remaining unenforced private functions and variables are monitored by run-time mechanisms which examine their call stack when they are invoked, and signal an error if they were not called through the appropriate public interface functions.

2.4 Subsystem Extraction.

A last example of structural evolution in exploratory development is the extraction of part of a system for use in a different context. For example, some set of functions developed for one system might provide behaviors closely related to the requirements for a different system. Successfully exploiting this form of community resource begins by answering the following "simple" question: Given the set of functions which implement the "top level" of the desired subsystem, what other functions and data types are required by this top level set?

In my proposed environment, the **extract-subsystem** process will provide automated support for developers answering this question. In traditional (non-exploratory) languages, this process would simply consist of static analysis of the system to determine the transitive closure of the calling hierarchy generated from the top level set. Interestingly, two characteristics of exploratory languages like Common Lisp—dynamic function invocation and polymorphism—make this process more complicated.

First, dynamic function invocation (through `funcall`, `eval`, and `apply`) make it difficult to place any statically-determinable constraints on the run-time calling hierarchy. Second, polymorphic functions can lead to the transitive closure mechanism generating a subsystem containing many extraneous functions not actually required in the new context. For example, a common form of polymorphic function calls different sets of functions depending on the type of object passed to it. For a particular interface set, only a subset of these types might be passed, resulting in a number of extraneous functions that the transitive closure technique would include.

Demand-based typing mechanisms again provide means to address these problems without imposing undue overhead on the development process. When the developer invokes the **extract-subsystem** process, she is prompted for the set of functions that will form the top level of the new subsystem. An augmented form of a transitive closure mechanism generates the subsystem. This mechanism queries the user for additional type information in the cases where dynamic function invocation occurs with insufficient constraints on its arguments to determine which functions should be included in the extracted subsystem.

The **extract-subsystem** process is also able to "rewrite" code in a limited sense. Suppose a given polymorphic function dispatches on the types of its arguments through a cond clause in the original system. In the extracted subsystem, however, suppose the polymorphism is reduced—fewer types will be passed, and thus some of the cond clauses will never be executed. When this situation can be recognized, the **extract-subsystem** will "extract" a new version of this function with the vestigial clauses removed. Not only does this result in clearer code, but also a smaller extracted subsystem, since the functions called within the deleted clauses will not be extracted.

Subsystem extraction is just the first step of a more far reaching process—the sharing of behavior and code between two systems. Improved support for this larger process is a possible direction of future research. For example, while the subsystem extraction process informs the environment that such sharing has begun, what further support can the environment provide downstream from this event? For example, should a copy of the code be created for the new system, or should a single instance of the code be shared? One form of such *software coevolution* is discussed next.

2.5 Software Co-evolution.

A virtually unexplored area in software development is software co-evolution, where a set of systems evolve in response to each other. One form of software co-evolution is a "extract-evolve-abstract" process, whereby a subsystem is extracted from one system for use in another, each version of the subsystem undergoes subsequent evolution within the two developmental contexts, and at a later time, an single abstraction of the two subsystems is created and used in both contexts.

Software co-evolution is a very common event in environments where a large number of exploratory projects are occurring in similar or identical development environments, such as the AI research in the University of Massachusetts COINS department. A typical example of software coevolution which I participated in occurred last spring between the Environment File Editor system [Hildum, 1988] and the Ada Restructuring Assistant system [Johnson *et al.*, 1988]. In this case, we discovered during the development of the ARA system that a suitable graphical interface for ARA could be similar to EFE's interface. So, we (manually) extracted the code relevent to the graphical interface from the EFE system, made a copy, and modified it for its new context. This process naturally led to a more domain-independent, more portable user interface code—at least for the version in the ARA system. The next, "natural" step—creation of a single interface module for both of these systems—was never taken for a single, simple reason: the high overhead of the process.

As implied above, the **extract-subsystem** process provides useful support for the first step of this **extract-evolve-abstract** process of software coevolution. Structural support for the subsequent evolution and abstraction involves representations that: (1) help the developers understand what structural modifications occurred as the two versions evolved independently within their separate contexts; (2) the structural differences that exist between the two versions of the subsystems at the time of abstraction; and (3) support for the structural "reconciliation" of the two versions into a common abstraction.

The representational mechanisms for software coevolution encompass not only the development of a single system, but the entire *community* of developing systems. In addition, software coevolution requires representation of code directly shared by two systems, and "code cousins"—sets of code with a common ancestor but incorporating adaptations to their own developmental context.

3 Environmental Support for Structural Evolution

While these examples of evolutionary processes promise aid in attaining structural fluidity, creating a supportive environment for their development and application poses new problems. First, each of these processes require structural representations at a variety of levels: from the level of entire projects sharing subsystems, to object and type aggregations, to the clauses making up a cond expression. Second, the processes aren't necessarily independent: subsystem extraction may lead to the aggregation of new objects or types, for example. Third, as indicated in the introduction, support for structural evolution must not only include mechanisms to automate change, but also mechanisms to explain how and why change occurred. In this section, I outline a key facet to be developed in my proposed research: the explicit representation and manipulation of software structure at multiple grain sizes.

3.1 A multi-grain size perspective on software structure

Software "structure" can refer to many things, such as the structure of a cond clause, the structure of an object or type hierarchy, the subsystem structure of a system, or the structure of a set of systems sharing subsystems directly, or even copying code and modifying it for their own purposes. I believe that environmental support for structural evolution depends crucially on the use of a representation for software structure that encompasses at least all of these meanings of structure. To my knowledge, no environment or language to date provides such a "broad spectrum" structural representation language, though separate, non-integrated mechanisms do exist for many of them. Here is a general overview of my work on such a representation language.

Character. The character grain size is the smallest grain size represented, since it normally corresponds to the smallest "building block" manipulated by developers in constructing source code.
Structural relationships at the character level consist of little more than successor and predecessor relationships.
Process phenomena also are limited to standard editor functions like insertion, deletion, formatting, and the like. The character level will be adapted from and built on top of the Explorer Zmacs editor environment.

Expression. Expressions are syntactically legal groupings of characters, which in Common Lisp correspond to S-expressions. Expressions can be recursively nested, such as when a cond clause is embedded within a **defun** expression. Top level expressions are not embedded within any other expression, and generally correspond to function and variable definitions.

Structural relationships at the expression level correspond to those recorded by cross referencing programs, such as calling relationships among functions, and referencing relationships among functions and variables.

Process phenomena concern the addition, deletion, and migration of expressions, which correspond loosely to compilation/evaluation (for addition), editor operations (for deletion) and a novel form of very fine-grained version control mechanism (for migration).

Object/Type. Sets of top level expressions are aggregated into objects and abstract data types. Classes and ADTs can be aggregated into larger classes and ADTs.

Structural relationships at the object/type level correspond to inheritance relationships between objects, and representations of public and private aspects of abstract data types.

Process phenomena concern events like object and type emergence, deletion, and extension, which were discussed above.

System. Sets of objects and abstract data types are aggregated into subsystems, which are themselves aggregated together into larger subsystems, and so forth until an aggregation representing an entire development project is represented.

Structural relationships correspond to PIC-style [Wolf, 1985] requires/provides information, as well as representations for code sharing and code "cousins" with other systems.

Process phenomena concern events like subsystem extraction, which was discussed above.

Community. Entities at the community level are entire development projects.

Structural relationships between projects represent shared subsystems, or subsystems with a common developmental ancestor.

Process phenomena concern events like the extract-evolve-abstract form of software coevolution, discussed above.

3.2 A multi-grain size perspective on software process

Within this framework, each evolutionary process "belongs to" one level, though it invariably triggers processes at other levels. For example, **extract-subsystem** is a system level process, through it might be initiated as part of a community level process to share two subsystems, and it may in turn eventually impact at the expression level by initiating a process to delete a cond clause.

In addition to supporting the decomposition of and interactions between processes, the multiple grain sized perspective allows the evolutionary history of the system to be viewed at different levels of abstraction. The environment not only represents the current state of the system, but also all previous states, and the evolutionary processes that produced one state from another. This corresponds in some ways to a version control facility, but contains significant new information about the structural events that transformed one version of the system into another. Traditional version control mechanisms do not store this information at any level of abstraction higher than the character level. While my proposed environment represents differences between versions at this level of abstraction, it *also* represents the differences between versions as the evolution of expressions, *and* as the evolution of objects and types, *and* as the evolution of subsystems, *and* as the evolution of relationships between this system and others in the community.

In addition, the decomposition of the subsystem extraction process into object/type and expression level processes means that developers can not only view the difference between successive versions of the system at the system level, for example, but they can also see how the evolutionary phenomena that occurred at the system level relate to the evolutionary phenomena at other levels.

Of course, evolutionary phenomena might not occur at all of these levels in every successive version of the system. Finally, the design of a user-friendly interface to the plethora of structural information stored in this environment is being left to post-proposal research.

4 Related research

4.1 Software structure evolution

4.1.1 Work at ISI

Research at the USC Information Sciences Institute on the Common Lisp Framework is closely related to this proposed research. While their environment is oriented primarily toward support for program synthesis, two papers focus on their approach to supporting structural change.

[Narayanaswamy, 1988] discusses the use of a static analysis tool embedded within the Common Lisp Framework [Goldman, 1988]. This tool determines typical cross referencing information (such as calls, uses, binds, sets, etc.) from analysis of the source code. It extends the capabilities of cross referencing systems by supporting "automation rules". These rules are production rules whose left hand side specify patterns to be matched against the current static state of the software. Whenever the cross referencing information about the software under development changes (such as through redefinition of a function, or addition of a new function), any applicable automation rule is invoked. The author cites applications such as automatic recompilation (such as when a list data structure is changed to a hash table), and various situations where the tool can inspect the changes made to a function and determine which other functions the programmer must modify as well.

While Narayanaswamy's paper focuses on expression level phenomena specific to Common Lisp, [Balzer, 1985] takes the same tack on object-level structures (a.k.a. "frame-based knowledge representation languages"). While the structural forms are different, and the supporting tools less sophisticated, the approach is identical: analyze the difference between two successive versions of the structure, and propagate the changes automatically if possible, or at least support the programmer in doing it themselves. Object dependencies are induced primarily through inheritance, and so the types of changes supported consist mainly of determining the effects of changes to the set of instance variables or methods of a particular class on all of its subclasses.

Both of these works share my emphasis on the use of structural knowledge to support evolutionary development. However, I believe that I take a more comprehensive approach to the representation and application of structural knowledge.

First, their work represents structure at a single grain size, corresponding to a collapsing together of my expression grain size and the object half of the object/type grain size. An obvious difference is thus my representation of structure at the type, system, and community grain sizes, which allows my environment to represent and support additional developmental phenomena.

A more subtle difference is the concept of grain size itself: that the structure of a system, as well as its evolution, can be viewed at these different levels. Thus, a developer can not only choose to view some portion of the system or its developmental history at, say, the level of the objects and types in the system, but can also "increase the magnification" by reducing the grain size to view those same changes at the expression level.

In addition, the static analysis tool reacts only to the differences between the current expression level structure and its immediate predecessor. Evolutionary phenomena that require a sequence of structural changes (such as object or type emergence) are not currently supported by their mechanisms.

Finally, their work raises a question: within a framework designed to support program synthesis (thus requiring higher level semantic specification of system function), why is this behavioral information not exploited to support structural change?

4.1.2 Pre-proposal research by the author.

Although I haven't always referred to it in that manner, most of my graduate school research relates to software structure evolution.

In the research reported in [Johnson and Lehnert, 1986], I designed an environment, the PLUMBER'S APPRENTICE, with "a set of languages which describe the developing system with different levels and types of abstraction." The structural focus of these languages clearly parallels and has influenced the current work. However, I felt the focus on natural language understanding systems to be artificially narrow: the phenomena, and the required mechanisms, appeared generic to any kind of exploratory development.

In [Johnson, 1987], I generalized my focus, and proposed the "type extension" development method for structural evolution of object oriented systems. This research addresses the same issue of this proposal: how to control and represent structural evolution, but through radical constraints on the development method, rather than through representation of knowledge about structure and process.

In two recent papers, I explored issues in the restructuring of software at the subsystem level. [Johnson, 1988] uses weak structural knowledge and a weak clustering algorithm to perform subsystem creation in an entirely automated fashion. The rather dismal results of that approach led me to design a system which used somewhat deeper structural knowledge in a collaborative modularization process between the system and the user. This work is described in [Johnson *et al.*, 1988]. However, the limitations of that system have convinced me that much greater performance can be achieved through the exploitation of structural representations at not only the subsystem level, but at other levels as well. My current research tests this belief.

4.2 Software development methods

By necessity, this research on structural evolution must at least "take a stand" of some sort on software development methods and the software process (and may ultimately be viewed *as* research on development methods and process).

The principle reason for a methodological stance is the fact that some methods view large-scale structural change as undesirable in or even absent from properly developed software. The classic "waterfall" software development method [Boehm, 1976] implies that structural change is a result of errors during pre-implementation life cycle phases. The "design for change" method [Parnas, 1979] takes a slightly more liberal view, permitting ADT representation re-implementation. However, this method still requires the types and extent of changes to be designed in prior to actual implementation.

More recent methods like "prototyping" [Bonet and Kung, 1984] and "spiral development" [Boehm, 1985] are even more permissive in the kinds of changes possible over the course of development. However, these changes tend not to occur through evolution of the original system, but through a succession of re-implementations.

In contrast to these approaches, exploratory development (aspects of which are described in [Sandewall, 1978; Swartout and Balzer, 1982; Sheil, 1984; Partridge, 1986]) views large scale structural change as acceptable, at least for experimental, domain embedded systems. Rather than a prescriptive method, exploratory development is a descriptive set of design heuristics such as structured growth. These heuristics are supported by certain language and environmental features such as: dynamic function and type definition; dynamic function binding and invocation; representation of code as a data structure in the language; and a blurred boundary between what functionality "belongs" to the environment and what belongs to the application[3].

The highly dynamic (both in the run-time and the evolutionary senses of the word) character of exploratory domains and development features come into direct conflict with conventional structural representation and enforcement strategies, which rely upon static analysis techniques. As a result, exploratory environments neglect to directly support some structural concepts, such as abstract data types, which have unquestionable benefits.

Thus, exploratory development is a fertile domain for exploration of structural evolution for two reasons: first, exploratory development exhibits and supports a wide variety of structural evolution, and second, the attempt to provide structural representations and support within its dynamic character provides new and interesting research issues.

4.3 The software process

Research on the software "process" differs from software development methods in the emphasis on the specific actions taken during development (such as: modules are compiled before executed), in contrast to the abstract strategies prescribed by methods (such as: step-wise refinement)[4]. My research can be viewed as a process-level approach to the description and support of the structural aspects of exploratory development. The plan-based approach [Huff and Lesser, 1988] and the process programming approach [Osterweil, 1987] exemplify much of the research.

The plan-based approach employs plan recognition techniques to recognize features of the development process from the pattern of tool invocations, such as direc-

[3]Not all of these elements are needed for exploratory development, of course.
[4]But see the process programming approach below.

tory and file creation and deletion, compiler invocation, etc. Once a plan has been recognized, planning techniques are applied to support the successful completion of the activity, by automatically carrying out certain tasks, or by detection and possibly correcting certain classes of errors. For example, the system can keep track of system versions, provide certain kinds of developmental summaries, and detect inappropriate tool invocation (such as editing a file "frozen" as part of a system release).

Process programming also supports certain evolutionary phenomena at this grain size. Process programming attempts to provide algorithms to implement the abstract strategies of development methods like step-wise refinement. A piece of a process program looks something like this:

```
For case := 1 to numcases do
 test(executable, tests[case]);
 if ~resultOK (result)
    then All_Fn_Perfectly_OK := False;
```

Process programming differs from the plan-based approach principally by its orientation towards top-down control of the process, while the plan-based approach offers both top-down control and bottom-up recognition of development. In addition, process programming currently remains a speculative research idea, with no reported experiences with an actual implementation.

My proposed research departs from both of these approaches primarily by the more comprehensive representations of structure and structural change. None of the evolutionary phenomena described in this proposal, for example, could be addressed by the current form of the plan-based or process programming approaches.

4.4 Type systems and type inference

My proposed demand-based typing technique adapts abstract data type definition constructs to the exploratory development environment through the use of type inferencing techniques. This adaptation requires certain trade-offs, which I will describe through comparison to other type systems.

All current abstract data type mechanisms are implemented within statically typed or strongly typed languages. Statically typed languages, such as Ada, require every variable and function in the source code to be assigned a type through static analysis. In languages like Ada, type assignment is normally done through explicit type declarations in the source code, though a small amount of type inference may be performed.

Strongly typed languages, such as FUN [Cardelli and Wegner, 1986] and ML do not require the types of all variables and functions to be identified explicitly. Instead, these languages require type conformancy—the ability to prove through static analysis that types are used consistently. In other words, rather than determine the exact type of an entity, these languages simply determine what constraints the use or implementation of the entity imposes on its type, and then proves that these constraints hold. Thus, for example, the `length` function in these languages takes an argument of type `list`(α)—where α is any type at all.

The major advantage of strongly and statically typed languages is that type inconsistencies are caught during compilation, eliminating the risk of these classes of errors escaping into the production environment. Second, little or no type checking needs to be performed at run time, allowing faster compiled code.

The disadvantage is that these mechanisms which *completely* check type information *statically* is that they exclude many expressive exploratory language features. First, constructs like `funcall`, `eval`, and `apply` are used to define and invoke functions and types at run-time—obviously problematic for static type mechanisms. Second, static mechanisms must represent the surface syntax of the language, yet constructs like the Common Lisp `macro` facility allow this syntax to be extended dynamically. Third, the equivalent representation of data and procedure in Lisp is problematic for type mechanisms: is

```
(lambda (x) (+ 1 x))
```

a list or a function? Obviously, it is both, yet no current type mechanism allows an object to be typed as both a data object and a function object at the same time. Finally, most object definition constructs rely upon run-time type determination and dispatching mechanisms.

My demand-based typing approach attempts to provide exploratory development environments with some of the robustness offered by these typing mechanisms, without disallowing the expressive power of exploratory language constructs. It accomplishes this by relaxing the completeness and static criteria of static and strong typing mechanisms. The completeness criteria is relaxed by not requiring all functions and variables to be typed, but only those whose types are necessary to enforce any ADTs defined in the system. Second, this enforcement need not be obtained solely through static means, but may extend into the dynamic environment if necessary.

Demand-based typing allows exploratory systems to be initially implemented with little or no type checking, but as the system evolves and "matures", increasing numbers of abstract data types can be introduced. Ultimately, an exploratory system could evolve to the point of using only abstract data types, which would provide much of the benefits of strong and static type checking mechanisms.

4.5 Related work in hierarchically evolving systems

My approach to software structure evolution is influenced greatly by research in other fields related on evolutionary phenomena, and the effects of evolution on system organization.

My multiple grain size representation of software structure corresponds to a "hierarchy of control" [Grene, 1987]. Hierarchies of control are "systems that consist of smaller units contained in larger ones, in such a way that the lower level units provide material for the arrangements at upper levels and the upper level arrangements constrain and thus control the activities of the lower levels". Polyani refers to these hierarchies as "systems of dual control"—while the activities at lower levels are constrained by the upper levels, the upper levels depend upon the lower levels for their very existence [Polyani, 1968].

Although understanding systems of dual control seems to involve only interactions at two levels, three adjacent hierarchical levels must actually be studied, since any single level is both constrained by the level above it and produced by the level below it. [Salthe, 1985] terms this the "triadic perspective", and claims it is essential to understanding any evolving hierarchical system.

The necessity of the triadic perspective is recapitulated by [Beer, 1985], this time in the context of business organizations. Beer views healthy business organizations as organized into a hierarchy of control, where at least some of the systems at each level must be "viable". Viable systems can exist in at least some environments other than the system in which they are currently found. In the context of business organizations, one viable system is a department that could be sold off and whose required goods and services could be obtained from an independent firm. Viable systems are necessary because it is only these that actually "produce" the system at the next level; the other, non-viable systems exist to stabilize the environment for the viable systems.

While these works make statements about the utility of hierarchical structure for complex systems, [Simon, 1962] goes even farther in claiming their necessity. In so doing, Simon introduces a subtly different constraint on component systems than Beer: rather than viability, he emphasizes stability, and supports this with the parable of Tempus and Hora. Tempus and Hora were two harried watchmakers whose assembly technique differed only by the presence of stable subassemblies. Hora produced 10 stable subparts of 10 pieces each, arranged ten of those into a stable subassembly of 100, and so on. When interrupted, only the last incompleted subassembly would fall apart. Tempus, on the other hand, had only one stable assemblage—the completed watch—and thus needed to start over each time he was interrupted. Recast in Beer's terms, Hora created a series of viable subsystems which could exist in an environment outside that of a completed watch, while Tempus's watch had no internal viable subsystems.

Simon's concept of stable subsystems has been applied to the domain of software development on at least two occasions. [Wegner, 1983] claims that the module construct is the software analogue of a stable subsystem, while [Fischer and Lemke, 1988] makes the same claim for objects and the inheritance hierarchies constructed from them.

I believe that these analogies miss the mark in an important sense, at least for exploratory development. Although stable subsystems are often reified through objects or modules, these structures do not comprise sufficient conditions for stability. The same logical error arises from claiming that molecules are stable subsystems for biological systems—although many types of molecules are stable, many others are not.

In exploratory development, stable subsystems can be constructed from modules or objects or types, but their crucial property is their stability in the face of change in the desired behavior for the system. Unstable subsystems can also be constructed from modules, objects, or types, but they end up either discarded or dismantled and reconstructed over the course of development.

As Tempus and Hora illustrate, stable subsystems catalyze evolution, speeding the generation of more complex behaviors. For this reason, stable subsystems are often created by decree, such as in the standardization of Common Lisp and other programming languages. At some point, the benefits obtained from stability outweigh the disadvantages of an imperfectly adapted language. This dynamic tension between stability and adaptation pervades exploratory development.

Beer's concept of viability relates to the desire for communal resource exploitation in exploratory development. For software resources to be shared between developing systems, they must be viable—able to exist within both environments. Mechanisms to identify viable subsystems can also be applied to the problem of team development: developers working on independent viable systems can more easily represent, understand, and control the effects of their modifications on the other's work. The **extract-subsystem** process provides some support for the creation of new viable subsystems.

Finally, Salthe's triadic perspective generates a viewpoint on how understanding (and thus representation) of the interactions between hierarchical levels could be organized. Change at one level of software structure can only occur when the structural context at the next higher level allows it, and has as an effect the inducement of structural change at the next level below it. The triadic perspective provides an organizing principle for the design of structural levels, evolutionary processes, and their interactions. For example, the **emerge-object** process must have as a precondition an appropriate system grain size organization, and must induce appropriate conformance at the expression level below.

The triadic perspective, stability, and viability are intriguing concepts to explore within my research, as my domain and environment provides the first opportunity to explore their computational encoding. However, I currently desire to leave this research avenue open for post-thesis efforts, and the next section outlines the chunk of work I intend for my immediate future.

5 Research plan

This research begins with the claim that frequent and unpredictable change is an inevitable consequence of exploratory development—the developmental strategy required for experimental, domain-embedded software. I believe that structural change is a significant component of the overall evolution of exploratory software, and that, furthermore, representation of structural knowledge can suffice to provide important new support for the process of exploratory development. What sets this research approach apart from much other research on software structure is its focus on the exploratory development process, and the implications of this process and its languages on the forms of software structure to be represented, and the types of support to be offered. Let me try to be more specific.

5.1 Contributions

This proposal outlines a research agenda that should yield the following contributions the subarea of artificial intelligence and software engineering:

- **Integration of structural representations.** Current structural representations stay basically within one grain size. By representing structure across several grain sizes, I can represent and (in some cases) automate the impact that changes to one grain size (such as the subsystem organization) make upon another grain size (such as the clauses in a cond clause). No other research to date has explored the structural knowledge and representations required for this form of integration. In addition, no other research has explored structural representations at the community level of abstraction.
- **Representation of developmental history at multiple (structural) levels of abstraction.** In exploratory development, it becomes much more important to understand how the system has changed, and to provide that information in an comprehensible manner. My multiple grain sized environment supports an automated documentation facility for display of the structural history at different

levels of abstraction. No other research to date has explored the use of structural information for automated documentation at different grain sizes.

- **Representation of ADTs within an exploratory environment.** Abstract data type facilities and the accompanying interface control mechanisms have long been acknowledged as important software engineering tools for the development of robust systems. However, these facilities have always been deployed within languages deemed unsuitable for the needs of most of the exploratory development community (i.e. AI software developers.) This research is the first to date to attempt to provide abstract data type facilities within an exploratory development environment.
- **A "demand-based" type system.** In service of the implementation of ADTs in an exploratory environment, this research hopes to contribute a new form of type mechanism which provides an interesting blend of features that bridges the relatively large gulf now separating strong and weakly typed langauges.
- **Automation of exemplar evolutionary phenomena at each grain size of structure.** Support for structural evolution is currently at the level of Narayanaswamy's static analysis system, which can only detect phenomena like changes to function arguments. While such facilities are undeniably useful and significant, the evolutionary phenomena I propose to implement, **emerge-object**, **emerge-adt**, **extract-subsystem**, and **extract-evolve-abstract** offer much more sophisticated support for structural evolution.

5.2 Timeline and the To-Do Set

Support for these contributions requires implementing the following parts of a prototype environment:

1. An integrated set of representations for expressing the structure of a Common Lisp program at the character, expression, object, type, system, and community grain sizes. The precise form of these representations will depend upon (2) and (3) below.
2. The demand-based typing mechanism (whose precise form will depend to some extent upon (1) above.)
3. The emerge-object, emerge-adt, extract-subsystem, and extract-evolve-abstract processes (whose precise form will depend upon (1) and (2) above.)
4. A system utilizing the multiple grain size structural representations to document the structural history of a Common Lisp program (whose precise form will depend upon (1) and (3) above.)

In addition, the thesis needs to be written. I expect to need about 6 months for programming, and 6 months for writing, which corresponds to a February 1990 graduation date.

5.3 Evaluation

I propose to evaluate my implementation through exercising each of the mechanisms in the following manner:

1. **The multiple grain sized representations.** These representations will be used to document aspects of the structure of certain exploratory systems in the department, such as the GBB system or the Environment File Editor. Interviews with the designers of the systems will be used to assess how well the representations express the structural relationships in these programs.
2. **The evolutionary processes.** These processes will be applied to the exploratory systems represented during the evaluation of the multiple grain sized representations, or to other representative system structures. The robustness and generality of these processes will be assessed through this experimentation.
3. **Demand-based typing.** This typing mechanism will be evaluated through analysis of its functionality. For example, can it function in the presence of any legal Common Lisp program, or are certain language constructs (such as macros) disallowed? How can the run-time performance penalties of this mechanism be described?
4. **Documentation of structural evolution.** The documentation mechanism will be evaluated by its utilization on a small example of exploratory development, and the (subjective) assessment of the utility of the levels of abstraction, as well as an analysis of how informative strictly structural history is for program understanding.

5.4 Future Directions

In researching general characterizations of structural evolution, I was struck by how well they apply to the process of exploratory development. While my thesis research uses this literature only to support my choice of a multiple grain sized architecture as appropriate for investigating evolutionary phenomena, I am very interested at some point in going the other way—attempting to utilize my framework to further these general concepts through their elucidation in a specific domain. For example:

[Salthe, 1985] claims that evolving biological systems are triadic systems. Triadic systems have two principle qualities: representation at multiple grain sizes and constraints upon inter-grain size interaction (such as grain sizes may only directly interact with their directly adjacent neighboring grain sizes.) Can triadic constraints be viewed as beneficial "design criteria" for the implementation of the processes of structural evolution such as type emergence, object emergence, and subsystem extraction? If not, what features of structural evolution invalidate the application of this model?

[Simon, 1962] claims that evolution is catalyzed by the presence of stable subsystems. Can stable software subsystems be identified or declared in the proposed environment? How can the concept of stable subsystems be exploited in exploratory software development?

[Beer, 1985] claims that organizations should be recursively composed of viable systems. Can viable software subsystems be identified or declared in the proposed environment? How can the concept of viable subsystems be exploited in exploratory software development?

6 Summary

Exploratory software development can benefit from better environmental support for structural change. Supporting structural change requires addressing a number of issues—What structural changes occur in exploratory development? What kinds of structural representations are needed? How can developers be helped to understand the types of changes that occurred? When is structural change appropriate? What are desirable and undesirable forms of structural change?

Fully answering these questions requires not only a deep understanding of the meaning of the software itself, but also of the motivations, goals, and relationships among the developers. However, even if this information were available, it would be of little use without support for the practical matters involved in actually changing the software structure. I believe that software to support structural evolution is experimental and domain-embedded: not only is its existence necessary to evaluate it, but its existence will also change the types of structural evolution that occur.

References

[Balzer, 1985] R. Balzer. Automated enhancement of KR systems. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, 1985.

[Beer, 1985] S. Beer. *Diagnosing the System for Organizations.* John Wiley and Sons, 1985.

[Boehm, 1976] B. Boehm. Software engineering. *IEEE Transactions on Computers*, C-25(12), December 1976.

[Boehm, 1985] B. Boehm. A spiral model of software development and enhancement. Technical Report 21-371-85, TRW, 1 Space Park, Redondo Beach, CA 90278, 1985.

[Bonet and Kung, 1984] R. Bonet and A. Kung. Structuring into subsystems: the experience of a prototyping approach. *Software Engineering Notes*, 9(5):23–27, 1984.

[Cardelli and Wegner, 1986] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4), December *1986*.

[Fischer and Lemke, 1988] G. Fischer and A. Lemke. Construction kits and design environments: Steps toward human problem-domain communication. *Human-Computer Interaction*, 3:179–222, 1988.

[Giddings, 1984] R. V. Giddings. Accomodating uncertainty in software design. *Communications of the ACM*, 27(5):428–434, May 1984.

[Goldman, 1988] N. Goldman. The common lisp framework: Reference manual. Technical report, USC/Information Sciences Institute, 1988.

[Grene, 1987] M. Grene. Hierarchies in biology. *American Scientist*, 75, September 1987.

[Hildum, 1988] D. Hildum. Graphical specification of DVMT environments. Master's thesis, University of Massachusetts, Department of Computer and Information Sciences, Amherst, MA, March 1988.

[Huff and Lesser, 1988] K. Huff and V. Lesser. A plan-based intelligent assistant that supports the software development process. In P. Henderson, editor, *Proceedings of the ACM Sigsoft/SigPlan Software Engineering Symposium on Practical Software Development Environments*, November 1988.

[Johnson and Lehnert, 1986] P. Johnson and W. Lehnert. Beyond exploratory programming: A methodology and environment for natural language processing. In *Proceedings of the National Conference on Artificial Intelligence*, 1986.

[Johnson *et al.*, 1988] P. Johnson, D. Hildum, A. Kaplan, C. Kay, and J. Wileden. An Ada restructuring assistant. In *Proceedings of the Fourth Annual Conference on Artificial Intelligence and Ada*, November 1988.

[Johnson, 1987] P. Johnson. Object orientation as a software engineering technique for artificial intelligence systems, 1987. Unpublished manuscript.

[Johnson, 1988] P. Johnson. Inferring software system structure. Technical Report 88-46, University of Massachusetts, Department of Computer and Information Science, April 1988.

[Letovsky, 1988] S. Letovsky. *Plan Analysis of Programs.* PhD thesis, Yale University, New Haven, CT., December 1988.

[Narayanaswamy, 1988] K. Narayanaswamy. Static analysis-based program evolution support in the common lisp framework. In *Proceedings of the Tenth International Conference on Software Engineering*, 1988.

[Osterweil, 1987] L. Osterweil. Software processes are software too. In *Proceedings of the Ninth International Conference on Software Engineering*, pages 2–13, 1987.

[Parnas, 1979] David L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, SE-5(2):128–138, March 1979.

[Partridge, 1986] D. Partridge. Engineering artificial intelligence software. *Artificial Intelligence Review*, 1(1):27–41, 1986.

[Polyani, 1968] M. Polyani. Life's irreducible structure. *Science*, 160, 1968.

[Salthe, 1985] S. Salthe. *Evolving Hierarchical Systems.* Columbia University Press, 1985.

[Sandewall, 1978] E. Sandewall. Programming in an interactive environment: The Lisp experience. *Computing Surveys*, 10(1), 1978.

[Sheil, 1984] B. Sheil. Power tools for programmers. In D.R. Barstow, H.E. Shrobe, and E. Sandewall, editors, *Interactive Programming Environments.* McGraw Hill, Inc., 1984.

[Simon, 1962] H. Simon. The architecture of complexity. *Proceedings of the American Philosophical Society*, 106, 1962.

[Swartout and Balzer, 1982] W. Swartout and R. Balzer. On the inevitable intertwining of specification and implementation. *Communications of the ACM*, July 1982.

[Wegner, 1983] P. Wegner. Varieties of reusability. In *ITT Proceedings of the Workshop on Reusability in Programming*, 1983.

[Wolf, 1985] A. Wolf. *Language and Tool Support for Precise Interface Control.* PhD thesis, University of Massachusetts, 1985.