Using Tool Abstraction to Compose Systems

David Garlan, Carnegie Mellon University
Gail E. Kaiser, Columbia University
David Notkin, University of Washington

Two complementary paradigms support the evolution of large-scale software systems. Data abstraction eases design changes in the representation of data structures, while tool abstraction does the same with system functions.

Managing complexity and supporting evolution are two fundamental problems with large-scale software systems.[1] Although modularization has long been accepted as the basic approach to managing complexity, as David Parnas observed nearly 20 years ago, not all modularizations are equally good at handling evolution.[2]

Data abstraction is a popular, important style of modularization. In this style, an abstract data type is defined by an explicit interface that specifies operations on instances of the data type. This approach defers design decisions about representing concrete data structures and implementing algorithms on those structures. These concrete decisions can be changed without modifying the module's clients, which are written in terms of the stable interface.

Enhancing a system's function typically accounts for about 60 percent of maintenance costs in a large system's life cycle, and hence roughly 40 percent of total software life-cycle costs.[3] If the interfaces to abstract data types are kept the same to protect clients from evolutionary changes, enhancements must often be constructed in terms of existing abstract data types. This restriction can lead to two problems. The desired function may not be computable from the existing interfaces, or implementing the function in terms of these interfaces may be unacceptably inefficient.

Thus, modifying the abstract interface itself may be the most effective — or the only — way to enhance functionality. Changing the abstract interface, however, implies that the concrete implementation must be understood and changed, which increases the complexity of the task. While one such change to an existing data abstraction may not be a serious problem, as the number of enhancements increases so does the complexity of the interactions between them.

Therefore, designers need an approach to handling changes that permits the system to be enhanced incrementally and modifications to be developed independently, even when the changes cannot be achieved by using traditional data-abstraction techniques. Several existing kinds of systems approximate these objectives. For example, spreadsheets are often enhanced by adding new equations that

0018-9162/92/0600-0030$03.00 © 1992 IEEE

use the values in data cells to interact indirectly with existing equations. In another example, production systems — a popular implementation paradigm for expert systems — consist of a collection of independent pattern-action pairs (called rules) that fire when the patterns match values in a shared database (called working memory). In principle, a production system can be enhanced by adding new rules that match and manipulate the working memory.

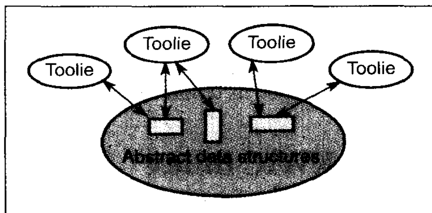


Figure 1. Collection of toolies that share a set of abstract data structures.

We call the diverse set of systems structured in this style the *tool-abstraction* paradigm. That is, despite some differences, each system of this sort has a common structure that encourages and eases incremental enhancement of system function, just as data abstraction encourages and eases changing design decisions about data representation. Rather than detract from the central idea of tool abstraction by introducing a new tool-abstraction mechanism, we use existing approaches to describe and argue the benefits of tool abstraction. The sidebars throughout this article describe several of these approaches.

Systems that support tool abstraction are structured as a pool of abstract data structures shared by a collection of cooperating "toolies," where each toolie provides a piece of the overall system function. When one toolie updates the shared data, other toolies must be notified; otherwise, cooperating-but-independent toolies may not execute, and the overall system function may be compromised. Figure 1 illustrates this architecture.

At this level, tool abstraction resembles trigger-based database-management systems that provide access to shared data through a common set of schemas. As discussed later, each system handles notification differently, although most use an event-based approach.

Tool abstraction complements, rather than supplants, data abstraction. Data abstraction allows design decisions about the representation of data structures to change easily, while tool abstraction allows system functions to change easily.

A simple example

Consider a small message system centered on a queue module that exports enqueue and dequeue operations only. Using abstract data types, how could we enhance the system so that it will not add duplicate messages? One approach is to modify the clients of the queue. However, this is unsatisfactory because to detect duplicates by using the original interface, the client would have to dequeue each message already on the queue, compare it to the new message to be enqueued, and then re-enqueue the original messages, plus perhaps the new one. Even though this activity could be encapsulated in a new client, the performance penalty is severe. Specifically, the number of enqueue/dequeue operations executed would be linear in queue length.

Another approach is to change the queue module implementation. The number of enqueue operations would drop to one, and no dequeue operations would be needed. However, this approach is unsatisfactory because the semantics (although not the syntax) of the enqueue operation must change. Thus, clients that want the original queue semantics (perhaps for some other use of queues) are not isolated from the change. Using data abstraction, this kind of problem must be handled by creating distinct abstract interfaces, each with different semantics.

Both approaches are less attractive in the face of multiple enhancements. Consider a second enhancement that adds a time stamp to each message; this necessarily interacts with the prohibition against duplicate messages. In particular, both enhancements must modify the implementation even though they may be conceptually independent. Further, the clients must usually be modified to gain any advantage from the time-stamp enhancement. Again, this is further complicated when various clients desire different semantics. Some clients might prefer to queue the dupli-

Spreadsheets

Spreadsheet programs have gained enormous popularity as flexible, extensible tools for financial accounting.[1] A spreadsheet can be viewed as a shared data pool represented by a matrix of values, with toolies represented by equations associated with positions in that matrix. When data in one of the matrix entries changes, the runtime system automatically reevaluates all equations that depend on that entry, updating the appropriate entries. Suppose an equation defines the rightmost value in a row as the sum of the row's other values. If a user changes one of those values, the spreadsheet will automatically reevaluate the sum. This in turn may trigger the reevaluation of other equations, such as one to add all values in the rightmost column.

A spreadsheet's toolie invocation mechanism thus depends on dataflow analysis to determine which matrix entries affect which equations. Many spreadsheets have simple mechanisms that do not handle, or even identify, circular relationships among the toolies. Luckily, in the domains most generally addressed by spreadsheets, such circularities rarely arise.

A spreadsheet system is not a general-purpose tool environment but a specialized application generator. Consequently, the range of toolies that a spreadsheet implementer can describe is constrained. Nonetheless, spreadsheets exhibit the architectural hallmark of tool abstraction: a shared pool of data together with event-driven control of function. Moreover, they effectively handle functional evolution: Circularities aside, equations can be added to the system independently of other equations in the system. While complex dependencies may exist between these equations, the system rather than the programmer manages those interactions.

Reference

1. R. Ross, *Design of Personal Computer Software*, IEEE Press, New York, 1985, pp. 282-300.

cate with the earliest time stamp, while others prefer to use the latest time stamp.

Although this example is especially simple, it illustrates some tensions that arise when systems evolve. The use of tool abstraction shows how problems of evolution can be reduced.

The message buffer serves as the shared data structure. One toolie provides the basic enqueue and dequeue operations. To handle the first enhancement, a "remove duplicates" toolie is defined; after the initiation of each enqueue operation, this toolie is invoked to compare the message about to be inserted with all other messages, aborting the enqueue operation if the message is already in the queue. Because it has direct access to the message buffer, the toolie can be implemented with reasonable efficiency. (If needed, the toolie can maintain an auxiliary representation of the buffer, keyed by whatever component of a message is checked for duplication.) The "add time stamps" toolie, which defines the second enhancement, is invoked when the enqueue operation terminates successfully. This toolie augments the newly inserted message with a field that represents the current time. Maintaining correctness is relatively easy, since each new toolie interacts only with the message-buffer representation and any other toolie invoked by the same operation, not with other toolies associated with the message queue. In this case, it's unnecessary to specify the invocation order of these two toolies, since "remove duplicates" is invoked upon initiation of enqueue, while "add time stamps" is invoked upon its termination.

The KWIC index production system

As Parnas pointed out, the issue is not *whether* to modularize a system — since modularization is essential to the control of complexity — but rather how to design the best criteria for decomposing a system design into modules. Parnas contrasted data abstraction with functional decomposition, showing that systems based on the former can better handle evolution for certain classes of change. We have taken this one step further. Tool abstraction reintroduces decomposition criteria according to function but uses a composition paradigm based on sharing abstract data structures (low-level abstract data types) among a collection of cooperating tools, together with an event-driven integration mechanism. A bundle of toolies represents a higher level abstract data type, amenable to a wide range of enhancements. This approach contrasts with the strict use of data abstraction and leads to better support for the more common classes of functional enhancements.

Parnas used the KWIC (Key Word in Context) index production system to compare two modularizations with respect to ease of evolution. To further illustrate the idea of tool abstraction, we do the same. The first design we present is Parnas's second decomposition, which uses data abstraction to decompose the system into modules. Our second design is based on the tool-abstraction paradigm. We show how tool abstraction (along with data abstraction) supports evolutionary enhancements more effectively than does data abstraction per se. Parnas describes the KWIC system as follows:[2]

> The KWIC index system accepts an ordered set of lines, each line is an ordered set of words, and each word is an ordered set of characters. Any line may be "circularly shifted" by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order.

As Parnas explains, this is a small system and consequently none of the motivating issues actually arise. Similarly, it's hard to properly evaluate evolution issues in a system of this size. However, because it does let us point out the key issues and problems of tool abstraction, we follow Parnas's lead in treating it as a large project that presents realistic problems.

Design 1: Parnas's decomposition. Parnas decomposes the KWIC system into modules that hide specific data-representation and algorithm choices so that these choices can be changed.

Production systems

This popular implementation paradigm for expert systems typically consists of a collection of rules, where each rule is a pattern-action pair. The pattern defines the conditions under which the associated action should be applied (or triggered). Patterns are written in terms of the values of objects in a shared database, called working memory. When several patterns apply to a particular database state, the system automatically controls the sequencing of corresponding actions, as determined by rule-ordering policies that vary with the kind of production system. These systems are based on tool abstraction, with working memory representing the shared data pool and the production rules representing the toolies. Interactions between the enhancement and existing functions are controlled by the production system itself.

The Formalized System Development (FSD) system[1,2] and Marvel[3] are rule-based software-development environment architectures that are relatively close to pure tool abstraction. Each combines ideas from production systems and active data, and represents the software artifacts under development in an object-oriented database. Tool fragments are automatically invoked in response to chaining on the rules, which in turn can be triggered by changes to the data. In FSD, new rules and subclasses of existing objects can be added at any time to accommodate additional Lisp tools. In Marvel, new rules, new classes of objects, and extensions to existing classes can be added at any time to integrate additional commercial off-the-shelf tools into the environment.

References

1. R. Balzer, "A 15-Year Perspective on Automatic Programming," *IEEE Trans. Software Eng.*, Vol. SE-11, No. 11, Nov. 1985, pp. 1,257-1,268.
2. D.S. Wile and D.G. Allard, "Worlds: An Organizing Structure for Object-Bases," in *Proc. SIGSoft/SIGPlan Software Eng. Symp. Practical Software Development Environments*, P. Henderson, ed., ACM, New York, 1986, pp. 16-26; and in Special Issue of *SIGPlan Notices*, Vol. 22, No. 1, Jan. 1987.
3. G.E. Kaiser, P.H. Feiler, and S.S. Popovich, "Intelligent Assistance for Software Development and Maintenance," *IEEE Software*, Vol. 5, No. 3, May 1988, pp. 40-49.

- The Line Storage module implements a sequence of lines, with routines to create, access, and delete characters, words, and lines.
- The Input module reads and stores the original lines.
- The Circular Shifter module provides routines to access individual characters, words, and lines of the circular shifts of the stored lines.
- The Alphabetizer module provides routines to access shifted lines in alphabetical order.
- The Output module prints the circular shifts in alphabetical order.

The top-level program first invokes the Input module, which stores the lines using the Line Storage module. The actual representation used by Line Storage is hidden from Input. Functions exported by the Circular Shifter module are then invoked. Circular Shifter retrieves the stored input using Line Storage, hiding decisions about data representation and algorithms from the top-level program. Then functions from the Alphabetizer module are used to sort the shifted data. Alphabetizer accesses the data through Circular Shifter, while hiding the sorting algorithm from the top-level program. Finally, the Output module accesses the sorted list using Alphabetizer (and perhaps Circular Shifter). In contrast to the common functional decomposition, alternative control structures are easy to construct.

Potential changes in the context of Design 1. Parnas's data decomposition is effective in handling such alternative representations and implementations as packed-versus-unpacked characters, explicit-versus-implicit representation of shifts, and monolithic-versus-incremental alphabetization. However, Parnas's decomposition does not directly support other kinds of enhancements. This is not surprising, since Parnas was concerned primarily with situations in which a system's functional specifications remained unchanged, although the implementations could vary. Looking at how some proposed enhancements might be supported in Parnas's decomposition of KWIC demonstrates some evolution problems in systems modularized according to the principle of data abstraction.

Consider augmenting KWIC with the capability to omit shifts that start with one of a set of noise words, such as articles. Parnas's decomposition admits two approaches to this modification.

The first one is to include a simple filter in the Output module (or add a small module that provides this filter). The filter checks the first word of each shift, printing the associated line only if it does not start with a forbidden word. This approach is straightforward but unnecessarily inefficient. In particular, all shifts must be alphabetized, including those that will ultimately be omitted from the output. This added cost can be significant, as can be seen by looking at the KWIC index for Unix manual entries, which is based on a one-line header associated with each Unix command. If this index listed all shifts, there would be about 5,000 entries. But the actual index omits shifts starting with about 150 noise words: Only about 1,000 shifts actually appear when the noisy ones are filtered out. Sorting dominates the overall cost. In this case, the cost is $O(N \lg N)$, where N is the initial number of entries, so this decrease in sortable entries saves at least a factor of five in execution time. This cost is indicative of the performance penalties that may be required by restricting access to encapsulated data.

The second approach to implementing the omit version of KWIC is to modify the Circular Shifter. (The Alphabetizer module could also be modified, but this approach is less attractive for similar concerns about performance.) As each call is made to the Circular Shifter to insert a new line, the line is checked against the set of prohibited words. If a match is found, the line is not inserted into the shifted list. The code to implement this approach is simple, and it keeps the structure of the shifter implementation straightforward. However, as we shall see, this kind of solution becomes complicated when further changes are considered.

A second possible enhancement closely relates to the first: Given a set of words, include only those shifts that start with one of them. This approach might be used to cull a set of smaller KWIC indices, each related to a subtopic.

The same implementation approaches are available, with the same tradeoffs. Filtering is easy, leaving the existing modules unchanged, but it is inefficient. Or the "include" check can be incorporated into the shifter, adding complexity to shifter code and raising the question of how to handle other shifter clients.

In this small example, modifying the shifter is not a serious problem. However, including both omission and inclusion enhancements in the shifter makes the module more complex. Rewriting it from scratch might produce a clean version, but it's not practical to rewrite a module each time an enhancement is made. So, in practice, module complexity tends to explode as repeated enhancements are made.

Two additional possible enhancements closely relate to the first two. In these, omission and inclusion are again provided, but on the original list of lines rather than on the shifts. These enhancements, when combined with the first two, help produce KWIC indexes that meet a wide range of needs. The same implementation trade-offs arise. Filtering can be done after the line storage is initialized, again at added cost (although not so bad as in the shifter's case, since the insertion cost is linear). Or the Line Storage module itself can be modified, just as the shifter was.

One problem with modifying the shifter and line-storage modules as suggested is that the decision to include a given line should not be the responsibility of those modules. This is not a serious problem for the simple enhancements we have discussed, but it becomes much more significant as other enhancements and modifications are introduced. For instance, what if a user wanted an enhancement where individual words could be included or excluded, as opposed to including or omitting shifts that start with these words? Implementing such changes efficiently in the line-storage or shifting modules would make the implementations confusing. Also, the capability to reuse modules (such as Line Storage) for storing other lists of words (such as the noise words) would be compromised because orthogonal enhancements might be needed for each list. In any case, module focus on line storage or shifting would fade, which is inappropriate because it compromises the separation of concerns. It should be possible to treat each enhancement as an independent unit, with the only interactions being through operations on the shared data structures.

Design 2: Tool abstraction. In Design 1, the enhancements — while possible — are unnecessarily complicated. In

particular, it's unfortunate that logically independent requirements are difficult to implement efficiently without intertwining logically independent implementations. For instance, deciding whether to include a shifted line should not affect the implementation of the shift module, but that's the most effective implementation in practice. Tool abstraction, in contrast, allows each toolie to act as a (largely) independent entity that focuses on a single function, decreasing the complexity of successive enhancements.

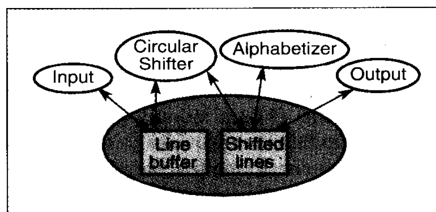


Figure 2. Toolies that share line-buffer and shifted-line abstract data structures.

With tool abstraction, the input, shifted, and alphabetized data are kept as shared (although still abstract) data structures. When a given toolie modifies the common data, other dependent toolies are invoked indirectly. This keeps each toolie clear of functionally unrelated code. The shared data are still abstract because they are not "opened" completely. In particular, the physical representations are still hidden by the data-abstraction mechanisms. The functions on these shared data entities are, however, factored into toolies. In contrast, traditional abstract data types combine these two concerns.

Design 2 is based on a collection of toolies that manipulate shared buffers representing sequences of lines. The following toolies define the basic operations of KWIC. Figure 2 illustrates their interaction.

- The Input toolie creates a new instance of a shared buffer. It reads lines from the input file, inserting each successive line into the shared line buffer.
- The Circular Shifter toolie creates another instance of a shared buffer to hold the shifted lines. It associates its action, in the sense of a *daemon* or *active data*, with the termination of the insert operation of the Input toolie. As each line is originally inserted, the shifter is implicitly invoked to create the shifts for that line. The Circular Shifter is not concerned with the internals of the Input toolie but only with its registered insert operation. (In practice, different systems handle registration in different ways; see the sidebars for examples.) Alternatively, the shift creation code could be associated with the complete input buffer, rather than with single lines, and therefore could be triggered when all input lines are inserted as the Input toolie signals completion. Thus, in this example at least, tool abstraction is suited for defining incremental and batch computations. The input lines and the shifted lines are conceptually separate buffers. Changing between creating a new buffer and using the input buffer directly, or between using an explicit or an implicit representation of the shifts, can be done by redefining the Circular Shifter toolie. This illustrates how toolies can augment data as well as function.
- The Alphabetizer toolie associates its action with the shared shift buffer. The Alphabetizer is triggered by the completion of shifter activities to sort lines in the buffer. In the case of an implicit shift buffer, this results in a coroutine interaction between the two toolies. Another shared buffer could be created to hold the alphabetized shifts, if desired, or the alphabetized buffer and the shift buffer could be equated to the same data structure. The sort could also be incremental, associating incremental insertions with the insertion of each shift into the shared shift buffer.
- The Output toolie provides a display scheme for printing the alphabetized shift buffer.

The top-level program invokes Input followed by Output. The actions of Input cause the Circular Shifter to execute, the shifter actions cause the Alphabetizer to execute, and Output simply accesses the sorted, shared buffer. An alternative approach would be for Alphabetizer to trigger an event when it's done sorting; Output would be invoked to print the results automatically.

Potential changes in the context of Design 2. Enhancements are accommodated more effectively and efficiently in this approach. Tool abstraction provides the capability to naturally define multiple enhancements independently of existing code, while still producing programs that execute efficiently.

An Omit toolie can be associated

Active data in object-oriented systems

Many object-oriented programming languages support some form of event-driven control, which is often used to update gauges and dials in user interfaces. Sometimes the language itself provides a notation to associate a method invocation with the changing value of an object's instance variable. In other object-oriented languages, event-driven control is provided as a set of kernel facilities. In Smalltalk-80,[1] for example, the system maintains a list of dependent objects for each object. An object can send itself the *changed* message, which causes an *update* message to be sent to each of its dependents. This mechanism underlies the Model-View-Controller paradigm[2] that drives many aspects of Smalltalk's user interface.

Another approach is exemplified by Flavors,[3] where methods inherited from multiple ancestor superclasses are combined by invoking method fragments before and after the main method. Most modern object-oriented databases also include some style of "trigger."

References

1. A. Goldberg, *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, Reading, Mass., 1984.

2. G.E. Krasner and S.T. Pope, "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80," *J. Object-Oriented Programming*, Vol. 1, No. 3, Aug./Sept. 1988, pp. 26-49.

3. D.A. Moon, "Object-Oriented Programming with Flavors," in *Proc. Object-Oriented Programming Systems, Languages, and Applications Conf.*, N. Meyrowitz, ed., 1986, ACM, New York, pp. 1-8; and in Special Issue of *SIGPlan Notices*, Vol. 21, No. 11, Nov. 1986.

with the insert operation (provided by the original Input toolie) on the shared shift buffer. If the first word of the shift being inserted is in the set of noise words, the toolie aborts the insert (that is, the Omit toolie is triggered by the initiation of the insert operation rather than its termination). The code for programs that insert into the shift buffer need not be changed. The cooperation between these toolies and the Omit toolie is triggered implicitly by operations invoked on the shared data rather than on explicit calls between operations. This eases evolution because multiple toolies can be triggered by the same operation without any changes to the triggering operation. The same approach holds for the Include toolie on the shift buffer. It also works for the Omit and Include toolies on the shared input buffer. When the input and shift buffers are made explicit, separate toolies can be associated with each one.

Structure-oriented environments

These environments consist of a collection of toolies that share structured (as opposed to textual) representations of program objects. Toolies in these environments typically share data represented as an attributed abstract syntax tree, whose form is defined by a grammar consisting of a collection of context-free productions.

Toolies are usually associated with these shared abstract syntax trees in the form of action routines or attribute grammars. An action routine is a procedure, associated with a production in a grammar, that defines the actions of some toolie on instances of that production. In the Gandalf system,[1] action routines are written in a special-purpose programming language called ARL (Action Routine Language), which is oriented toward manipulating abstract syntax trees. Each action routine is associated with an operation on an abstract syntax tree node (such as create or delete, which indicates when it should be invoked). An alternative to action routines is to associate a set of attribute equations with each production in the grammar, as done in the Synthesizer Generator.[2]

Roughly speaking, an attribute equation defines the value of an attribute of an abstract syntax tree node as a function of the attribute values of adjacent nodes in the tree. Attribute equations determine a graph of data dependencies that can be used to incrementally reevaluate attributes when an editing toolie changes the shared tree.[3]

In both approaches, control is event driven. In the first case, toolies written as action routines are automatically activated when data is manipulated by the operation associated with the routine. In the second case, toolies written as attribute equations are automatically invoked when updates are made to the data on which the attribute values depend. Enhancement is simplified in both cases, since typically the new function can be added as new action routines or attribute equations that are logically distinct from the existing ones.

References

1. A.N. Habermann and D. Notkin, "Gandalf: Software Development Environments," *IEEE Trans. Software Eng.*, Vol. SE-12, No. 12, Dec. 1986, pp. 1,117-1,127.
2. T. Reps and T. Teitelbaum, "Language Processing in Program Editors," *Computer*, Vol. 20, No. 11, Nov. 1987, pp. 29-40.
3. T. Reps, *Generating Language-Based Environments*, MIT Press, Cambridge, Mass., 1984.

Discussion

Tool abstraction relates to a variety of other concepts.

Tool abstraction versus pipes. Toolies are similar in intent to Unix pipes, which link together a preplanned sequence of small functional units. Each unit in the pipeline takes as input the output produced by the previous unit; execution is triggered by the arrival of this input. Pipes are much more limited than toolies because pipes must be connected sequentially, while toolies share abstract data and are permitted to have much richer control interactions. A unit cannot react to operations on the data made by subsequent units in the pipe. Furthermore, in contrast to toolies, units connected by a pipe share only a single predefined data representation — a sequence of characters — with no capability for shared data definitions. When this is not a suitable representation for the internal processing of a particular unit, the unit must parse its input from and unparse its output to the standard character stream form. In contrast, each bundle of toolies can define its own data representation, with later enhancements perhaps adding components to the data structure.

These two restrictions do not pose a problem for the previously suggested enhancements to KWIC. For instance, pipes are a natural (although not particularly efficient) solution to omit/include on original/shifted lines. The input simply streams through a pipelined sequence of sorters and filters, and more filters can always be inserted in the pipeline. In other situations with more complex interactions among functional units, pipes are inadequate. The trade-off is not simple. The restrictions on pipes can be viewed as a way of managing complexity, but with the restrictions comes a reduction in the kinds of systems that are easy to build.

Tool abstraction versus inheritance. Inheritance in object-oriented languages can be used to provide some aspects of tool abstraction. In particular, inheritance is an especially good approach to extending abstract data types. In many cases, a subclass can provide additional operations on an existing data type without modifying the base type.

However, inheritance doesn't provide all the features needed for tool abstraction. The most notable exception is events. These can be added to object-oriented systems (as with the Smalltalk-80 Model-View-Controller[4]), but inheritance doesn't do the job by itself. Also, when inheritance is used to achieve code sharing rather than behavior sharing, the relationship to tool abstraction is even less clear. Handling triggering effectively is especially difficult. Perhaps more fundamentally, most object-oriented systems do not encourage programming in the paradigm of tool abstraction, even when they provide many of the underlying mechanisms.

Additionally, inheritance imposes a hierarchy on data abstractions. Toolies, in contrast, are equals. In particular, one could define a system that uses a subset of existing toolies, picking and choosing from desired functions. This is not straightforward with inheritance, where

Other systems

The paradigms considered in previous sidebars briefly sampled the kinds of systems based on tool abstraction. Other examples include monitors in the Domain Software Engineering Environment,[1] views and rules in relational database systems,[2,3] some extensions to the Interface Description Language,[4,5] and artificial intelligence blackboard architectures.[6]

One of the major goals of the RPDE3 software development environment is "to support both the integration of tools constructed from many small fragments and the construction of tools that can be extended to process new types of data without source-code changes."[7] The most significant limitation of RPDE3 with respect to tool abstraction is its lack of event-driven tool invocation. Tool interleaving is instead carried out by using explicit message-passing and routing rules.

The INC programming language has been developed for writing incremental computations.[8] INC is based on the observation that "there are many problem domains that require repeating a computation many times, each time on slightly different data," and the claim that the specialized techniques for efficient recomputation developed for such domains can be unified in a language-based solution. INC program components are thus tiny toolies, with event-driven integration specified by a circuit graph of components.

Even some systems that do not support tool abstraction take a similar approach with regard to opening up the representation of data to a variety of computational entities. In Famos (Family of Operating Systems), for instance, different levels of virtual machines can share representations of abstract data types by having a module extension "join" a module to access its internal representation.[9]

References

1. D.B. Leblang and R.P. Chase Jr., "Computer-Aided Software Engineering in a Distributed Workstation Environment," in *Proc. SIGSoft/SIGPlan Software Engineering Symp. Practical Software Development Environments*, P. Henderson, ed., ACM, New York, 1984, pp. 104-112; and in Special Issue of *SIGPlan Notices*, Vol. 19, No. 5, May 1984.

2. H.F. Korth and A. Silberschatz, *Database System Concepts*, McGraw-Hill, New York, 1986.

3. M. Stonebraker, E.N. Hanson, and S. Potamianos, "The Postgres Rule Manager," *IEEE Trans. Software Eng.*, Vol. 14, No. 7, July 1989, pp. 897-907.

4. D. Garlan, "Extending IDL to Support Concurrent Views," *SIGPlan Notices*, Vol. 22, No. 11, Nov. 1987, pp. 95-110.

5. R. Snodgrass and K. Shannon, "Supporting Flexible and Efficient Tool Integration," in *Advanced Programming Environments*, Vol. 244 of *Lecture Notes in Computer Science*, R. Conradi, T.M. Didriksen, and D.H. Wanvik, eds., Springer-Verlag, Trondheim, Norway, 1986, pp. 290-313.

6. L.D. Erman et al., "The Hearsay-II Speech Understanding System: Integrating Knowledge to Resolve Uncertainty," *ACM Computing Surveys*, Vol. 12, No. 2, June 1980, pp. 213-253.

7. W. Harrison, "RPDE3: A Framework for Integrating Tool Fragments," *IEEE Software*, Vol. 4, No. 6, Nov. 1987, pp. 46-56.

8. D. Yellin and R. Strom, "INC: A Language for Incremental Computations," *Proc. SIGPlan 88 Conf. Programming Language Design and Implementation*, ACM, New York, 1988, pp. 115-124; and in Special Issue of *SIGPlan Notices*, Vol. 23, No. 7, July 1988.

9. A.N. Habermann, L. Flon, and L. Cooprider, "Modularization and Hierarchy in a Family of Operating Systems," *Comm. ACM*, Vol. 19, No. 5, May 1976, pp. 226-272.

selecting a subclass implies selecting the properties of its superclasses.

Finally, inheritance can be viewed as a model for merging toolies into a cohesive system. This is the approach taken in the Meld language.[5]

Events. Our examples rely on using an event (or trigger) mechanism. The basic reason is that triggers allow greater independence among toolies.[6] Without triggers, toolies would be directly responsible for cooperatively managing their collective control flow. This process might be simple at first but would become increasingly complicated as enhancements were introduced.

Triggering can be implemented as an implicit, underlying mechanism or as an explicit, programmer-level mechanism. In attribute grammar-based structure-oriented environments, for instance, designers of specific environments are unaware of a triggering mechanism. They simply write the desired set of attribute equations. The same is true for spreadsheet users. But the underlying implementation triggers incremental evaluation when values are updated by the user. In production systems, the programmer is aware of how rules fire and must build systems based on these semantics.

When triggering is used as an implicit mechanism, many difficulties are managed by the underlying system. Circularity, for instance, can be a serious problem. Toolies can indirectly invoke themselves, producing an unbounded execution. Most attribute grammar and spreadsheet systems, however, check (either statically or dynamically) for such circularities, so their designers need not be overly concerned about circularities of an attribute grammar or a spreadsheet. In action routine-based structure-oriented environments, however, the programmer must be aware of this problem. The difficulties of trigger-based programming can be significant. The benefits, however, are also significant. There is now a wealth of practical (although perhaps not systematically understood and documented) experience with trigger-based programming, from such domains as structure-oriented environments, production systems, recent relational and object-oriented databases, object-oriented systems, and access-oriented programming (as in Loops[6]).

Efficiency. We have addressed several dimensions of efficiency in this article.

One dimension is the question of algorithmic complexity of the underlying system. Abstract data types, and their associated implementations, are generally designed to efficiently support a set of operations. If the interface remains inviolate, it may no longer support new functions efficiently as the system evolves. In the example discussed in the introduction, checking for duplicates by using the simple enqueue/dequeue interface is an instance of this problem; the fixed interface makes what ought to be a constant-time operation into a linear-time operation (in terms of the number of invocations on the interface). In some cases, the complexity is the same, but the constant can increase significantly, which may not be acceptable in practical systems.

Another dimension is whether triggering mechanisms slow down systems too much. There are at least two issues here. First, designers have quite a bit of experience with triggering mechanisms in a wide variety of domains (see sidebars). Many realistic systems have been based on triggering mechanisms, so there seems to be no inherent hurdle to overcome. Second, any added costs would be a matter of a constant factor, since the operation would have to be invoked anyway. With triggers, it's just a question of haggling about the price. This contrasts with the algorithmic dimension, since restricted interfaces can slow programs down by greater-than-constant costs.

Table 1. Instances of tool abstraction.

System	Shared Data	Toolies	Control Mechanism
Spreadsheets	2D matrix	Equations	Dependency analysis
Structure-oriented environments	Abstract syntax tree	Action routines	Operations
Structure-oriented environments	Abstract syntax tree	Attribute equations	Dependency analysis
Production systems	Working memory	Rules	Patterns and rule resolution
Object-oriented systems	Object base	Methods	Active values

Language issues. Parnas's initial paper presented the notion of data abstraction largely separate from language and implementation issues. In fact, data abstraction can be practiced without special language features if the programmers are sufficiently disciplined. With tool abstraction, however, language and runtime support are necessary because implicit invocation of toolies eases incremental evolution. This is not a serious problem, however, since the systems described in the sidebars provide concrete evidence that tool abstraction can be efficiently realized (although perhaps in restricted domains).

There can be no absolute judgment about which criteria and integration mechanisms are best. A structuring technique that's appropriate in one circumstance may be inappropriate in another. System structures based on the functional decomposition criterion criticized by Parnas may be appropriate when physical data formats are unlikely to change, but new paths for processing existing data are expected as the system evolves. Witness the success of Unix pipes. On the other hand, decomposition based strictly on data abstraction may be appropriate when data representations are likely to change, but not when external interfaces are also likely to change. Finally, as we have argued, composition by using tool abstraction is appropriate when a system evolves through enhancements to the function supported by existing data structures.

Although our examples have been especially simple, functional enhancement does happen in complex situations. Consider the complexities that arise when basic telephone services are enhanced with such services as call forwarding and call waiting. For example, if call forwarding is turned on and the phone is in use, should the call-waiting tone sound when an incoming call arrives or should the call be forwarded? In each case, data from the same sensors must be processed, the same billing data updated, and so forth. While the message queue and KWIC examples could reasonably be reimplemented from scratch when an enhancement occurs, this is not feasible for most phone system features. Thus it becomes especially important for the implementation of optional new telephone services to be as independent as possible of the implementation of plain old telephone services.

Luckily, it's not necessary to choose one abstraction paradigm over the others. This is partly because no hard boundaries exist between these abstraction techniques. Parnas's decomposition of KWIC based on data abstraction appears very much like the functional decomposition. The difference is that his interfaces hide representations of data and other implementation decisions. Similarly, tool abstraction is not antithetical to abstract data types: Indeed, the data pool shared by a collection of toolies can itself be an abstract data type. Moreover, it's reasonable to expect that a collection of toolies would be bundled as an abstract data type that hides the details of a particular decomposition and that enhancements would be made by adding toolies to the bundle. The Meld language takes this approach.[5]

What, then, is to be gained from all this? The abstraction paradigm or paradigms that drive a system's design should be chosen to reflect the system's expected evolution. If functional enhancements are expected to be the primary form of change (as they are in most large systems), tool abstraction represents a particularly attractive and realizable addition to existing approaches.

This has ramifications for three classes of software engineers. *Language implementers* need to incorporate tool-abstraction principles into new language designs and develop general implementation techniques. *Environment builders* need to develop facilities that support system design based on tool abstraction (and analysis, testing, and debugging aids that support adding new toolies to existing systems). Finally, *system designers* need to be aware of the tool-abstraction paradigm and actively seek a system that supports shared data and event-driven tool integration. As explained in the sidebars, many instances of tool abstraction are widely used (also see summary in Table 1).

The tool-abstraction paradigm raises several interesting and difficult research problems. When multiple toolies react to the same event, ordering becomes an issue. There is also the potential problem of circularities among the dependencies implied by toolie events. When events are a programming-level paradigm, how can toolie independence be retained while circularities are prevent-

ed? Another problem is how to handle lazy-versus-eager invocations of toolies. For applications concerned with meeting timing constraints, how can the time costs of indirect invocation be understood and managed?

We have discussed toolies only in the context of the kinds of data structures typically encapsulated in abstract data types. But tool abstraction might support events other than operations on shared data structures. For example, it should be possible to attach toolie invocation to lack of data (when or where data is expected) and other exceptional conditions, to timer interrupts and other signals, and to events accessible only through polling some external entity (such as sockets and sensors). Because software systems involving these kinds of events are typically quite complex, the tool-abstraction paradigm should prove particularly fruitful. ■

Acknowledgments

We thank Bill Griswold, Ralph London, Chip Maguire, Josephine Micallef, Harold Ossher, Steve Popovich, Kevin Sullivan, Michael van Biema, Travis Winfrey, and Ursula Wolz for their detailed comments on drafts of this article. Nancy Griffeth suggested the telecommunication example.

Garlan is supported by National Science Foundation Grants CCR-9109469 and CCR-9112880, by a grant from Siemens Corporate Research, and by DARPA Grant MDA972-92-J-1002.

Kaiser is supported by National Science Foundation Grants CCR-9000930, CDA-8920080, and CCR-8858029, by grants from AT&T, Bell Northern Research, Citicorp, Digital Equipment Corp., IBM, Software Research Associates, Sun Microsystems, and Xerox, by the Center for Advanced Technology, and by the Center for Telecommunications Research.

Notkin is supported in part by National Science Foundation Grant CCR-8858804, Air Force Office of Scientific Research Contract AFOSR-88-0023, and grants from Digital Equipment Corp. and Xerox.

References

1. F.P. Brooks Jr., "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer*, Vol. 20, No. 4, Apr. 1987, pp. 10-19.

2. D.L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Comm. ACM*, Vol. 15, No. 12, Dec. 1972, pp. 1,053-1,058.

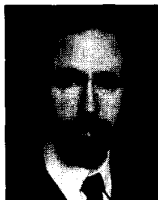
3. B. Lientz and E. Swanson, *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*, Addison-Wesley, Reading, Mass., 1980.

4. G.E. Krasner and S.T. Pope, "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80," *J. Object Oriented Programming*, Vol. 1, No. 3, Aug./Sept. 1988, pp. 26-49.

5. G.E. Kaiser and D. Garlan, "Melding Software Systems from Reusable Building Blocks," *IEEE Software*, Vol. 4, No. 4, July 1987, pp. 17-24.

6. M.J. Stefik, D.G. Bobrow, and K.M. Kahn, "Integrating Access-Oriented Programming into a Multiparadigm Environment," *Computer*, Vol. 19, No. 1, Jan. 1986, pp. 10-18.

7. K.J. Sullivan and D. Notkin, "Reconciling Environment Integration and Component Independence," *Proc. ACM SIGSoft 90: Fourth Symp. Software Development Environments*, ACM, New York; and in Special Issue of *SIGSoft Notices*, Vol. 15, No. 6, Dec. 1990, pp. 22-33.



David Garlan is an assistant professor of computer science in the School of Computer Science at Carnegie Mellon University. His research interests include the application of formal methods to the construction of reusable software architectures, programming environments, tool integration, and interactive maps. He has recently been active in developing formal models of embedded instrumentation software and in building environments to support the development of these models.

Garlan received the BA from Amherst College, the MA from the University of Oxford, and the PhD in computer science from Carnegie Mellon University in 1987. He is a member of the IEEE Computer Society.

Gail E. Kaiser is an associate professor of computer science at Columbia University. She was selected as a National Science Foundation Presidential Young Investigator in Software Engineering and was awarded a Digital Equipment Corp. Incentives for Excellence award, an IBM Research Initiation Grant, and several AT&T Foundation Special-Purpose Grants. Kaiser's research interests include software-development environments, testing and debugging tools, cooperative transaction models, reusability, application of artificial intelligence technology to software engineering, object-oriented languages and databases, and parallel and distributed systems.

Kaiser received the ScB degree from the Massachusetts Institute of Technology in computer science and engineering, and the MS and PhD degrees in computer science from Carnegie Mellon University. She is a member of the AAAI and the ACM, and is a senior member of the IEEE.

David Notkin is an associate professor in the Department of Computer Science and Engineering at the University of Washington. During 1990-91, he was a visiting faculty member at the Tokyo Institute of Technology and at Osaka University. In 1988, he received a National Science Foundation Presidential Young Investigator Award. His research interests include software engineering, environments, and evolution.

Notkin received the ScB degree from Brown University in 1977 and the PhD degree in computer science from Carnegie Mellon University in 1984. He has served on numerous program committees and is a charter member of the editorial board of the ACM *Transactions on Software Engineering and Methodology*. He is a member of the IEEE Computer Society and the ACM, where he is secretary/treasurer of SIGSoft.

Readers can contact Garlan at Carnegie Mellon University, School of Computer Science, 5000 Forbes Ave., Pittsburgh, PA 15213; Kaiser at Columbia University, Department of Computer Science, 500 W. 120th St., New York, NY 10027; and Notkin at the University of Washington, Department of Computer Science and Engineering, FR-35, Seattle, WA 98195.