# Dynamic Contextual Reflection: A Mechanism for Software Evolution and Reuse

**(Position Paper)**

Robert J. Walker and Gail C. Murphy

Department of Computer Science
University of British Columbia
201-2366 Main Mall
Vancouver, BC V6T 1Z4
Canada
{walker, murphy}@cs.ubc.ca

**Abstract.** Current approaches to programming cause external information to be encoded into components. When this information is not of importance to the implementation of these components, but is an artifact of programming mechanisms, system structure suffers, resulting in greater difficulties in software evolution and reuse. We are investigating an approach to lessen the effects of such *extraneous embedded knowledge* by reflecting upon dynamic execution information and static structural information, which comprise the concept of *context*.

## 1 The Problem

Experience has shown that abstraction is fundamental to supporting the development of complex systems. Abstraction permits certain ideals for the software life-cycle. Software designers can specify large-scale structure, leaving the details to be filled-in by implementors. Software implementors can focus their attention on the details of a restricted subset of a system while maintaining a high-level view of the rest. Software maintainers can modify a subset of a system without the need to alter the rest.

In practice, each ideal is rarely achieved, and never without the explicit intent to support the other stages of the software life-cycle. Designers need to be concerned with low-level details, because low-level interactions can impinge upon high-level structure. Implementors need to be concerned with the way data structures are used and interactions occur, not in isolation, but throughout a system. Maintenance activities are curtailed to prevent the need to propagate changes throughout a system; this results in structural degradation.

The abstractions provided by the design and programming languages in use today are not sufficient to meet the goals of each stage in the software life-cycle. Current approaches to design and programming cause too much external information to be encoded into components. When this information is not of

importance to the implementation of these components, but is an artifact of design or programming mechanisms, system structure suffers, resulting in greater difficulties in software evolution and reuse. We refer to knowledge of the external world that is not explicitly required for the behaviour of a component as *extraneous embedded knowledge* (EEK).

Consider the implementation of an object-oriented class `A`. This class is considered to be a structural unit in many respects. Any code not contained within `A` should have no knowledge of the details of implementation of the methods of `A`; such external code does contain explicit knowledge of the interface to `A`. Methods within `A` contain explicit knowledge of the interface to `A`, the interface to external classes, and the details of implementation of other methods within `A` to a degree.

There are a number of problems with this situation. The interface to `A` cannot change without breaking client code of `A`. The interface to classes external to `A` cannot change without breaking code within `A` (and other classes). Protocols requiring methods of `A` to be called in particular sequences are not explicit within the interface, and so, can be violated accidentally or maliciously.

Furthermore, the behaviour of `A` is relatively inflexible. If `A` must serve a multitude of clients, it must present a uniform appearance to them. This means that the methods of `A` are too constrained: they could be generic, presenting bland, unspecialized services, or they could possess complex sets of parameters allowing properties to be specialized. An unspecialized service often does not meet the needs of its clients; an overly-configurable service is prone to incorrect usage and is harder to change. Both cases arise as a consequence of EEK: a specific signature or configuration protocol should not generally be of importance to a client—save that the resulting communication needs to work.

EEK comes in many forms. The simplest of these is dependence on particular names and signatures of external components. It should be a simple matter to have a mechanism that could alias messages, replacing one name with another, or that could reorder parameters.

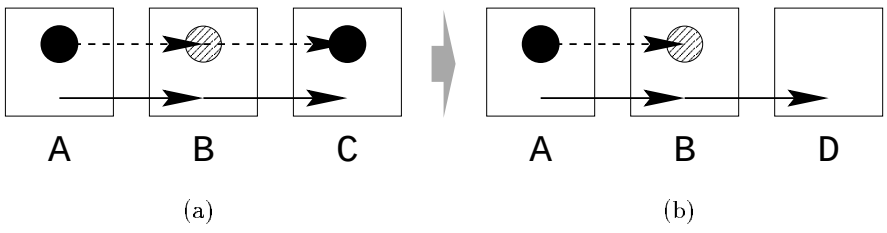


Fig. 1: Method `C` is replaced with method `D`, which does not need the parameter sent from method `A`, but the data-flow from `A` still passes to `B`, which does not use it. The solid arrows indicate control-flow, the dotted are data-flow.

As an example of more complex EEK, consider three methods: `A`, `B`, and `C`. Method `A` calls `B`, and `B` subsequently calls `C` (see Fig. 1a). In these calls, various parameters are passed; among these is a piece of information called `snip`. Method `C` requires `snip` for its execution and `A` is in the best position to obtain or calculate `snip`. Method `B` does not use `snip` in any way except to pass it on to `C`. At some point, it is decided that `C` should be replaced within `B` by a new method, `D`. Method `D` serves the same purpose as `C`, but does not require that `snip` be passed to it (see Fig. 1b). Since we do not want to break all of `B`'s clients, we do not change the interface to `B`—it still requires that `snip` be passed in to it, a parameter that it has no use for.

The disparity between the external functionality expected by a component, and the actual external functionality present in a given system containing that component must be overcome. The need for early binding of names and the fragility of encapsulation in interface protocols cause EEK to arise. Current design and programming mechanisms introduce these restrictions; a new approach is needed to reduce the influence of EEK.

## 2 The Approach

Coupling between components can be mitigated, making them more reusable and easier to change, by reducing or eliminating the extraneous embedded knowledge within them. Such a reduction is possible through extensions to the concepts of reflection and dispatch.

Reflection is ordinarily defined in terms of monitoring and altering what is currently occurring within a system—not what has already taken place. Many attempts have been made to leverage the idea of "context" in interpreting messages or selecting implementations [13, 15, 10]. These approaches are quite static, looking only at the current state of the system, or more likely, some small portion thereof. But the previous state and execution of the system has a lot to say about what should happen next: whether certain components have been used yet when they need to have been, or which library should be used in conjunction with servicing a message from a particular object. Just as in human speech, we can use statements and concepts from earlier communication to understand current requests, and we can modify our responses according to whom we are speaking and under what circumstances. As long as messages do not become ambiguous, we can be more concise, providing only that information that is really necessary.

More concretely, consider the problem of extraneous parameters again (see Fig. 1), where component `C` requires `snip` from `A`, and it happens to be passed through `B` because that is where the control-flow goes. Since `snip` is extraneous to `B`, it is needed by `B` only because of language constraints—the logical service provided by `B` does not suggest a need for `snip`. Therefore, `snip` should bypass `B` altogether (see Fig. 2). When the control-flow arrives at `C`, `snip` should be "filled-in" from the context of what has happened previously in the system.

When the message from `B` arrives at the boundary to `C`, our mechanism might look for the most recent object of `snip`'s type that had previously been

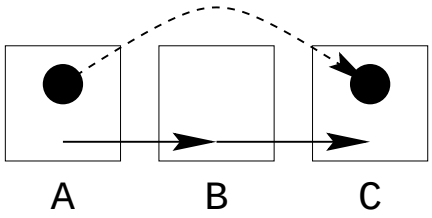


Fig. 2: The data-flow from method `A` bypasses `B`, which is not interested in it, thereby eliminating the EEK from `B` that would have been otherwise present.

passed—and perhaps not received—and fill in the appropriate parameter to `C`. Or it might look for the name `snip` and do the filling in that way. To make this safer than dynamic scoping, the identity of the component or pathway providing `snip` could be checked against. As an alternative, the filling-in could occur at the boundary to `B` to operate on outgoing messages there.

To permit arbitrarily late binding of names, such message manipulation occurs at the boundary of a component before the message enters the component, for incoming messages, or before the message exits the component, for outgoing messages. All the messages passing out of that component can be rerouted arbitrarily at this boundary; likewise, in-bound messages can be screened and rerouted here. This means that the code within the component need not be aware of the true names and interfaces of external components, and the internal code does not need to know of the deception. Parameters can be reordered, removed, or added at these boundaries through dynamic contextual reflection. Furthermore, to increase dynamic flexibility, the selection of a component to route a message to can also be dependent upon the history of execution of the system. A single apparent message could be serviced by multiple calls within a component, or multiple messages could be stored up until enough information were available to perform a single operation. To perform such message manipulations requires there to be descriptions at the boundary of a component describing which messages should be intercepted and how they should be manipulated when intercepted; we call these descriptions *boundary maps* (see Fig. 3).

To allow *dynamic contextual reflection* on the call history, a record must be kept of what has happened in the system thus far. Ideally, one can record the complete tree of calls as it occurs, replete with the objects sending and receiving messages and the objects being passed within those messages. Queries can then be made against this structure, with its mix of temporal and structural relationships, to garner information to alter new messages as they occur.

It is infeasible to record every method call with attendant objects and maintain this information over the entirety of the execution of any non-trivial system. Therefore, optimizations and limitations will need to be introduced to make the mechanism practical.

Combining contextual querying with the structuring and dispatch mechanism should permit the necessary flexibility to adapt components and systems to

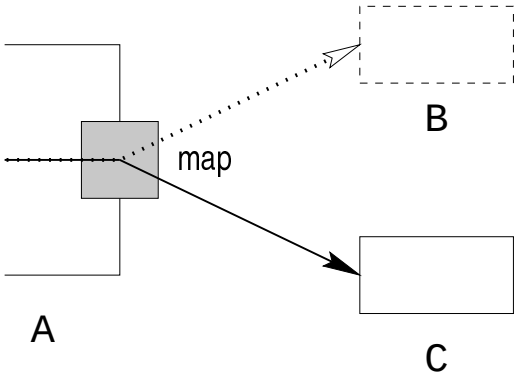


Fig. 3: An outgoing message is intercepted at the boundary of component `A` by a boundary map. The message contents can be manipulated and the message rerouted to a new recipient (`C`), all based on the previous history of the system. The originally intended recipient (`B`) need not be present in the system at all.

new situations. By moving the bindings of component interactions from within components to their boundaries, the components should be more easily reusable and the system more easily evolvable.

## 3 Related Work

No existing mechanisms address all forms of EEK simultaneously.

Global variables are a standard means of sharing information without passing parameters. There are several standard objections to the proliferation of global variables, including name-space collisions and violation of encapsulation [17]. Every component accessing a global variable is strongly dependent on its name and type, increasing component coupling and the presence of EEK.

Implicit invocation (a.k.a. publish-subscribe, event multicast) [6] is a means of separating control-flow from explicit knowledge of the names of components. Implicit invocation can remove some EEK arising from the knowledge of the names of subscribing classes and methods, but much remains: all components in an implicit invocation protocol relationship (callback registrar, subscribers, and event publisher) need to be aware that this particular mechanism is in place, subscribers and event publishers need to recognize a common interface for passing events and what those events are, all probable sources of EEK.

Predicate classes [3] are a generalization of multiple dispatch [2] that permit the type of an object to be transiently redefined according to its state (or according to a user-defined predicate that can be fairly arbitrary). Context relations [15] provide a language-based mechanism in support of the Strategy pattern [5] by allowing "context objects" to be dynamically attached to instances. Subjectivity [7, 9] allows different method implementations to be executed for a message depending on the run-time type of the sender of the message. Such a

mechanism could provide flexibility in interpretation of names within messages, but would still require too restrictive an agreement on the meaning of those names. All three of these mechanisms permit significant dynamic flexibility, and hence might address the need for eliminating early binding, but they do not provide any special means for eliminating the forms of EEK not arising from early binding, such as extraneous parameters.

Subject-oriented composition [14] is a means for composing and integrating disparate class hierarchies (subjects). Hence, the meanings of particular names can be rebound in composition specifications, effectively producing late binding. This will not remove EEK within each subject though.

Dynamic scoping (e.g., in Lisp) allows names to be bound into an external, non-lexical scope at run-time. This is notoriously fraught with evolutionary problems, as there is no guarantee that identical names in different scopes will be semantically equivalent. Even if they initially are semantically equivalent, an intervening scope can later be introduced with a non-equivalent variable name. Quasi-static scoping [11] allows explicit dynamic scoping only, removing the worst hazards arising from evolution; however, it provides no means for flexible dispatch of messages. Generic programming [12] is similar in that one essentially defines required interfaces to dummy classes that are then bound (typically via a template mechanism) to produce real implementations, but this static operation does not address dynamic needs, nor produce the data-flow separation required.

Context reflection [13] allows interpretation of messages and knowledge in terms of an explicitly-set, current context, allowing late binding, but providing no means for eliminating other problems leading to EEK.

Behaviorally adaptive objects [10] separate objects into two separate, interacting entities: crystals to represent the state of an object, receive messages, and select behaviour, and contexts to define operations. If more than one context is appropriate for the response to a message, the crystal must explicitly order the behaviours it selects and somehow resolve conflicts between them. Contexts are defined across sets of crystals too, tightly coupling them as a result. Behaviorally adaptive objects are fraught with EEK—even more than other approaches due to the tight coupling of crystals.

Composition filters [1] permit messages to be remapped much as boundary maps do. However, no reflection on system history occurs. Furthermore, while the filters themselves are reusable, they are used by specifying them explicitly within classes, resulting in poor separation of concerns, greater EEK, and less reusable classes.

Traces [8] allow the interpretation of messages to be altered based on a limited form of dynamic context. An explicit list of "ancestor classes" may be attached to an object; methods may be interpreted differently depending on whether the ancestor list of the receiving object matches pre-specified lists. Such ancestor lists can be thought of as particular paths through the call history tree, but at a coarser granularity than methods. Thus, traces permit a limited means of reflecting upon system history. However, since traces provide no means of

obtaining objects related to the history, it is not possible to apply traces to the problem of extraneous parameters described earlier.

LambdaMOO [4] and Perl [16] both permit access to the current call stack, but to no other, prior calls. Neither provides a means for retrieval of passed parameters.[1]

## 4 Progress and Conclusion

We have presented a brief description of the problem of extraneous embedded knowledge—a problem that limits the evolvability and reuse of software components—and suggested a potential solution that lies in reflecting upon the dynamic call history of a system. The call history is explicitly recorded as a tree, and queries can be made against this structure. Messages are intercepted at component boundaries and remapped on the basis of the call history with its attendant objects.

We are in the midst of implementing a dynamic contextual reflection mechanism. It operates by instrumenting Java source code to both record call history information, and to perform message remapping.

## References

[1] Mehmet Akşit, Lodewijk Bergmans, and Sinan Vural. An object-oriented language–database integration model: The composition-filters approach. In Ole Lehrmann Madsen, editor, *European Conference on Object-Oriented Programming*, volume 615 of *Lecture Notes in Computer Science*, pages 372–395. Springer-Verlag, 1992. (ECOOP '92; Utrecht, The Netherlands; 29 June–3 July).

[2] Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. CommonLoops: Merging Lisp and object-oriented programming. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 17–29. ACM Press, 1986. (OOPSLA '86; Portland, USA; 29 September–2 October). Published as ACM SIGPLAN Notices 21(11), November 1986.

[3] Craig Chambers. Predicate classes. In O. M. Nierstrasz, editor, *ECOOP '93—Object-Oriented Programming*, volume 707 of *Lecture Notes in Computer Science*, pages 268–296. Springer-Verlag, 1993. (1993 European Conference on Object-Oriented Programming; Kaiserslautern, Germany; 26–30 July).

[4] Pavel Curtis. *LambdaMOO Programmer's Manual*, March 1997. Version 1.8.0p6. ftp://ftp.lambda.moo.mud.org/pub/MOO/ProgrammersManual.ps.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, USA, October 1994.

[6] David Garlan and David Notkin. Formalizing design spaces: Implicit invocation mechanisms. In Søren Prehn and W. J. (Hans) Toetenel, editors, *VDM '91: Formal Software Development Methods, Volume 1: Conference Contributions*, volume 551 of *Lecture Notes in Computer Science*, pages 31–44. Springer-Verlag, 1991. (4th

[1] Except, in Perl, when the `caller` function is called within the DB package.

International Symposium of VDM Europe; Noordwijkerhout, The Netherlands; 21–25 October).

[7] William Harrison and Harold Ossher. Subject-oriented programming: A critique of pure objects. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 411–428. ACM Press, 1993. (OOPSLA '93; Washington, USA; 26 September–1 October). Published as ACM SIGPLAN Notices 28(10), 1 October 1993.

[8] Gregor Kiczales. Traces (a cut at the "make isn't generic" problem). In Shojiro Nishio and Akinori Yonezawa, editors, *Object Technologies for Advanced Software*, volume 742 of *Lecture Notes in Computer Science*, pages 27–43. Springer-Verlag, 1993. (First JSSST International Symposium on Object Technologies for Advanced Software; ISOTAS '93; Kanazawa, Japan; 4–6 November).

[9] Bent Bruun Kristensen. Subjective method interpretation in object-oriented modeling. In *Proceedings of the 5th International Conference on Object-Oriented Information Systems*. Springer-Verlag, 1998. (OOIS '98; Paris, France; 9–11 September).

[10] Stefan M. Lang and Peter C. Lockemann. Behaviorally adaptive objects. *Theory and Practice of Object Systems*, 4(3):169–182, 1998.

[11] Shinn-Der Lee and Daniel P. Friedman. Quasi-static scoping: Sharing variable bindings across multiple lexical scopes. In *Conference Record of the Twentieth ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 479–492. ACM Press, 1993. (POPL '93; Charleston, USA; 10–13 January).

[12] David R. Musser. Introspective sorting and selection algorithms. *Software—Practice and Experience*, 27(8):983–993, August 1997.

[13] Hideyuki Nakashima. Context reflection. In Akinori Yonezawa and Brian C. Smith, editors, *Proceedings of the International Workshop on New Models for Software Architecture '92: "Reflection and Meta-level Architecture"*, pages 172–177, 1992. (IMSA Workshop '92; Tokyo, Japan; 4–7 November).

[14] Harold Ossher, Matthew Kaplan, Alexander Katz, William Harrison, and Vincent Kruskal. Specifying subject-oriented composition. *Theory and Practice of Object Systems*, 2(3):179–202, 1996.

[15] Linda M. Seiter, Jens Palsberg, and Karl J. Lieberherr. Evolution of object behavior using context relations. *IEEE Transactions on Software Engineering*, 24(1):79–92, January 1998.

[16] Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., Cambridge, UK, second edition, 1996.

[17] W. Wulf and M. Shaw. Global variable considered harmful. *SIGPLAN Notices*, 8(2):28–34, 1973.