In *Proceedings of the 21st International Conference on Software Engineering* (16–22 May 1999, Los Angeles, CA, USA).

Contextual Programming

(Doctoral Symposium—Extended Abstract)

Robert J. Walker
Department of Computer Science
University of British Columbia
201-2366 Main Mall
Vancouver, BC, Canada V6T 1Z4
+1 604 822 3061
walker@cs.ubc.ca

ABSTRACT

When information external to a component is not of importance to the implementation of that component but is present within it as an artifact of design or programming mechanisms, system structure suffers, resulting in greater difficulties in software evolution and reuse. I am investigating an approach to lessening the effects of such *extraneous embedded knowledge* through the use of dynamic execution information and static structural information, which comprise the concept of *context*.

1 THE PROBLEM

Current approaches to design and programming cause external information to be encoded into components. When this information is not of importance to the essence of the these components but is an artifact of design or programming mechanisms, system structure suffers, resulting in greater difficulties in software evolution and reuse. I refer to knowledge of the external world that is not explicitly required for the specification of a component as *extraneous embedded knowledge* (EEK). EEK comes in many forms; space does not permit a full recitation.

As an example of EEK, consider three methods: `A`, `B`, and `C`. Method `A` calls `B`, and `B` subsequently calls `C` (Figure 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` (Figure 1b). Method `D` serves the same purpose as `C`, but does not require that `snip` be passed to it. Since we do not want to break all of `B`'s clients, we do not change the interface to `B`. Our system now performs work that is unneeded and conducts communication that is unwanted; aside from inefficiency, the code in support of this EEK obscures the meaningful operations within these methods. The system is harder to change and is more likely to contain errors.

EEK arises because of the early binding of names by programmers and the fragility of encapsulation in interface protocols. Current design and programming mechanisms necessitate these restrictions; a new approach is needed to reduce the influence of EEK.

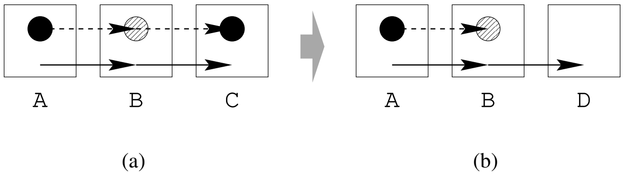

Figure 1. *Method `C` is replaced with method `D`, which does not need the parameter sent from method `A`, but the data-flow from `A` still passes to `B`, which does not use it. The solid arrows indicate control-flow, the dotted are data-flow.*

2 PRIOR RESEARCH

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 [6]. Every component accessing a global variable is strongly dependent on its name and type, increasing component coupling and the presence of EEK.

Predicate classes [1] 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 [5] provide a language-based mechanism in support of the Strategy design pattern by allowing "context objects" to be dynamically attached to instances. Subjectivity [2] 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 (Figure 1).

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.

Reflection [4] can allow a program to monitor and alter itself dynamically. However, reflection is a general principle that could be used by other mechanisms in mitigating the effects of EEK and not a mechanism in itself.

Behaviorally adaptive objects [3] 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.

Many other related mechanisms exist, but space does not permit their description. No existing mechanisms address all forms of EEK simultaneously.

3 THE APPROACH

Coupling between components can be mitigated, making them more reusable and easier to change, by reducing or eliminating the EEK 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 (e.g., [5, 3]). 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 have 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 which is really necessary.

More concretely, consider the problem of extraneous parameters again, 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 (Figure 2). When the control-flow arrives at `C`, `snip` should be filled in from context. The context mechanism might look for the most recent object of `snip`'s type that was "floating in limbo"—in context, that is—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.

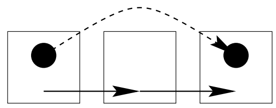

Figure 2. *The data-flow from method `A` bypasses `B`, which is not interested in it, thereby eliminating the EEK from `B` that would have otherwise been present.*

Combining data-flow separation with a particular structuring and dispatch mechanism should permit the necessary flexibility to adapt components and systems to 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. With the elimination of extraneous constraints and code arising from too much knowledge about particular components, their interfaces, and protocols, components should be cleaner to write and more closely represent their core concern.

ACKNOWLEDGEMENTS

This work would not have been possible without the help and encouragement of Gail Murphy.

REFERENCES

[1] C. Chambers. Predicate classes. In *ECOOP '93—Object-Oriented Programming*, pages 268–296, 1993. LNCS 707.

[2] W. Harrison and H. Ossher. Subject-oriented programming: A critique of pure objects. In *Proc. OOPSLA '93*, pages 411–428, 1993.

[3] S. Lang and P. Lockemann. Behaviorally adaptive objects. *Theory and Practice of Object Systems*, 4(3):169–182, 1998.

[4] P. Maes. Concepts and experiments in computational reflection. In *Proc. OOPSLA '87*, pages 147–155, 1987.

[5] L. Seiter, J. Palsberg, and K. Lieberherr. Evolution of object behavior using context relations. *IEEE Trans. on Software Engineering*, 24(1):79–92, 1998.

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