Using Implicit Context to Ease Software Evolution and Reuse

Robert J. Walker and **Gail C. Murphy**
Department of Computer Science
University of British Columbia
201-2366 Main Mall
Vancouver, BC V6T 1Z4
Canada
+1 604 822 5169
{walker, murphy}@cs.ubc.ca

Technical Report TR-99-13

11 November 1999

Abstract

Software systems should consist of simple, conceptually clean components interacting along narrow, well-defined paths. All too often, this is not reality: complex components end up interacting for reasons unrelated to the functionality they provide. We refer to knowledge within a component that is not conceptually required for the individual behaviour of that component as extraneous embedded knowledge (EEK). EEK creeps in to a system in many forms, including dependences upon particular names and the passing of extraneous parameters. This paper proposes implicit context as a means for reducing EEK in systems. Implicit context combines a mechanism to reflect upon what has happened in a system through queries on the call history with a mechanism for altering calls to and from a component. We demonstrate the benefits of implicit context by describing its use to reduce EEK in the Java Swing library.

Keywords
Software structure, flexibility, call history, contextual dispatch.

1 Introduction

When we begin building a software system, we typically strive to design components that are simple and conceptually clean. When we finish building a version of the system, a different story has typically unfolded. An original vision of independent and cohesive components that interact along narrow paths is too often replaced with a reality in which there exists a larger than desired set of interactions between components.

Obviously, components must communicate to provide system behaviour. Communication leads to interaction between components. The problem resides in the fact that a component ends up interacting with other components for reasons not directly related to providing its behaviour. For example, when a class participates in the Abstract Factory pattern [7] as a client, it must be aware of this participation; the abstract factory class must be explicitly named even though only the product classes managed by the factory are of interest to the client.[1] Such explicitly-named interactions make software brittle. We refer to knowledge of the external world within a component that is not conceptually required for the individual behaviour of that component as *extraneous embedded knowledge* (EEK).

A possible solution lies in the way humans speak to each other. Humans do not spell out every concept they wish to communicate at every instance the understanding of those concepts is required. We expect much information to be understood from or altered by context. Such use of context takes two forms: *elision*, where words are left out to be filled-in from either cultural knowledge or earlier details within a conversation, and *modification*, where the words that are spoken or the way that they are interpreted depends upon the individuals who are speaking. "It spun wildly" could refer to a ride at the county fair, or to one's impression of a room while experiencing extreme nausea; the details about "it" have been elided, to be understood from what has previously been discussed. Likewise, one's response to the question, "What does politics mean?" might be quite different depending on who is asking; the explanation given a young child is likely to be significantly modified from that given an adult.

Analogously, we can use elision and modification to simplify components and to reduce the coupling between them. When a message is passed or received, additional details (such as parameters) can be filled-in by reflecting upon what

[1] We use the term *component* to refer to a structural unit such as a method, class, or module when we do not care to differentiate between these.

has been previously said—the *call history* within the system. Furthermore, a message can be altered depending on to whom it is being sent or from whom it is being received; that is, messages can be intercepted before or after being sent and be replaced by other messages depending on the context in which they occur. We call this *contextual dispatch*. Combining these two concepts gives us a powerful mechanism, *implicit context*, for removing EEK from components.

To demonstrate the approach, we present a proof-of-concept application of implicit context to the 1,304-class Java™ Swing graphical user interface library. We show how the use of implicit context helped to make components in Swing simpler and less brittle. We were able to apply implicit context incrementally, evolving parts of Swing to use implicit context while running side-by-side with unchanged components.

We begin by expanding upon our description of EEK and giving a concrete example of its presence within the Swing library (Section 2). We then describe a mechanism for recording and utilizing implicit context, and explain its application towards reducing the presence of EEK within Swing (Section 3). In Section 4, we discuss issues that arise in using implicit context and in providing automated support for the approach. Section 5 compares implicit context to other related approaches. Finally, we summarize our arguments and findings in Section 6.

2 Extraneous Embedded Knowledge

Extraneous embedded knowledge creeps into a component as the component is elaborated and implemented. Sometimes a developer recognizes when EEK is infiltrating a component: this situation may be marked by a programmer exclaiming, "To make this work, I suppose I must link to the...". More often, EEK silently invades a component.

A complete categorization of EEK is not warranted for this initial investigation of implicit context. Instead, we focus on a few of the more common forms of EEK. We postpone a discussion of how existing approaches address EEK until Section 5.

2.1 Forms of EEK

The simplest form of EEK is the dependences a component, say `C`, forms on particular names and signatures of external components. If any of the external names or signatures change, component `C` will break. What should be important to `C` is not *who* will be providing desired external functionality, but rather *what* functionality is needed.

A more subtle version of EEK arises between components. Consider three methods: `mA`, `mB`, and `mC`. Method `mA` calls `mB`, and `mB` subsequently calls `mC`. In these calls, various parameters are passed; among these is a piece of information called `param`. Method `mC` requires `param` for its execution and `mA` is in the best position to obtain or calculate `param`. Method `mB` does not use `param` in any way except to pass it on to `mC`. At some point, it is decided that the call within `mB` to `mC` should be replaced by a call to `mD`. Method `mD` serves the same purpose as `mC`, but does not require that `param` be passed to it. One option is to change the interface to `mB`, but this would break all its clients. Instead, `mB` is typically stuck accepting a parameter for which it has no use. From the perspective of `mB`, `param` is an extraneous parameter.

EEK also arises in the form of protocol adherence. Consider a class `Cls` that has (at least) two methods: `init()` and `doit()`. `Cls` requires that the `init()` method be called prior to any calls to `doit()`. There are a couple of ways to handle this constraint, as follows.

We can force all clients of `Cls` to ensure that `init()` is called prior to `doit()`; however, since no client can be assured that it will be the first one to have called `doit()`, a flag needs to be set to indicate when `init()` is called. Such a flag is likely to be stored as a global variable or as a field within `Cls`, say `isInitialized`. Every client of `Cls` must then recognize and correctly adhere to this protocol to check and set `isInitialized`, spreading this concern everywhere.

Alternatively, we can force `doit()` to call `init()` the first time it is itself called. Although this approach also requires a flag, this flag is internal to `Cls`. This situation still makes `Cls` too brittle. The `doit()` method must know about `init()` to correctly follow the protocol. If the protocol is altered, the implicit dependence of `doit()` must be recognized and `doit()` must be modified appropriately. If `doit()` is modified, the developer must be careful not to introduce protocol violations.

With only two methods to be concerned with, this seems like a trivial problem. But with hundreds of methods to manage within a class, or with protocols that involve multiple classes, evolution becomes difficult and dangerous.

2.2 The Java Swing Library

As an example of where EEK arises and how implicit context can address EEK, we describe a part of the Java Swing library. Swing is a graphical user interface (GUI) toolkit that is intended to provide consistency in GUI appearance across platforms and to make it easy to build sophisticated widgets. Swing is distributed as part of Sun Microsystems's JDK 1.2.

A major feature of Swing is its *pluggable look-and-feel* (PLAF) architecture [6]. This architecture allows the look-and-feel of a GUI to be altered dynamically. As an example, a user interface in the Motif look-and-feel can be altered at run-time to a Windows look-and-feel. We will focus on the EEK associated with the part of the PLAF architecture that supports the changing of look-and-feels. We begin with an overview of the architecture followed by some (simplified) details of how the architecture works. The details are necessary to recognize the EEK.

2.2.1 Overview

In Swing, each GUI widget object contains a separate object, called a *UI delegate*, which is responsible for the display and interactive characteristics of the widget for a particular PLAF. For example, a class `JButton`, which implements a button widget, has an associated class `ButtonUI`, which provides its look-and-feel; `ButtonUI` has a separate subclass for each different look-and-feel. When `JButton` receives a message to paint itself, it forwards the message to its installed UI delegate, say a `MotifButtonUI` object, which draws the button properly according to its current state. When the look-and-feel of a widget is to be changed, the current UI delegate object for that widget must be uninstalled, the new UI delegate class must be located and instantiated, and the new UI delegate object must be installed on the widget.

2.2.2 How PLAF Works

Classes representing GUI widgets should be simple since GUI widgets themselves are conceptually simple. In Swing, these classes are complex, containing many details that are needed to support the PLAF architecture as well as other features, such as a separable model architecture. The `JButton` class, for example, defines or inherits a total of 183 public methods within the `javax.swing` package, plus 144 public methods from within the `java.awt` package—all just to implement a button!

Figure 1 shows a partially stripped-down object interaction diagram for the process of locating, instantiating, and installing a new UI delegate into a `JButton` object.[2] There are six classes involved in this process.

- `JButton` is a GUI widget for which the look-and-feel is to be changed.

- `BasicButtonUI` is a specialized button UI delegate. This class inherits from `ButtonUI`, which provides a generic base class for button UI delegates.

- `BasicButtonListener` is an event handler that responds to events, such as button presses, in a PLAF-specific manner. It is explicitly installed onto a given button widget by a button UI delegate.

- `LookAndFeel` is a base class for the various PLAFs. Each subclass of `LookAndFeel` specifies the set of UI delegate classes that are appropriate for its look-and-feel. Each class has an associated string—a `uiClassID`—that describes its purpose. For example, the `MotifLookAndFeel` specifies that `MotifButtonUI` corresponds to the `"ButtonUI"` purpose and that `MotifRadioButtonUI` corresponds to the `"RadioButtonUI"` purpose.[3]

- `UIDefaults` is used by `LookAndFeel` and its subclasses to store the mappings from the `uiClassID`'s for a PLAF to the actual UI delegate classes.

- `UIManager` is an abstract class with various static methods for registering the `UIDefaults` information for the current PLAF.

[2]The diagram ignores details concerning applet contexts for multiple applications running in the same virtual machine, as well as various initialization steps.

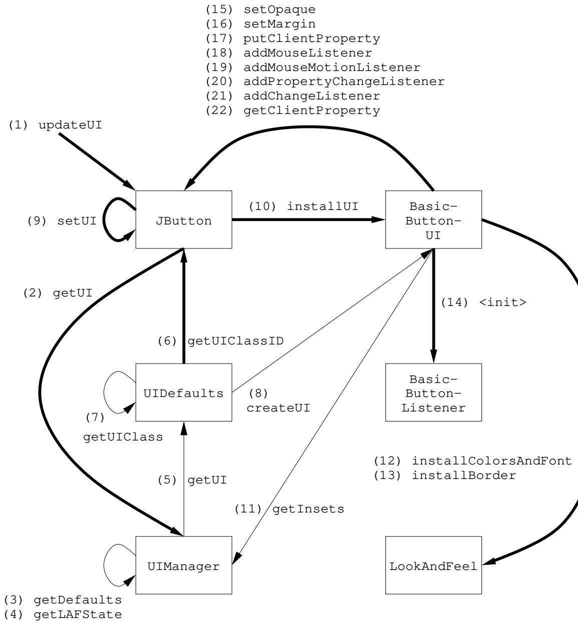


Figure 1: *Object interaction graph for the process of installing a "Basic" PLAF UI delegate into a* `JButton`*.*

The interactions between these six classes to support the changing of the look-and-feel are complex. Figure 1 depicts the over 20 messages involved. The interactions represented describe what happens right after the look-and-feel has been changed via a method call to the `UIManager` class. At that point, the application must explicitly call a utility method to run around and invoke each widget's `updateUI()` method. For `JButton`, this results in a request to `UIManager` to obtain a UI delegate object that is appropriate to the new PLAF. `UIManager` passes the current PLAF and the widget asking to be updated to `UIDefaults`. `UIDefaults` asks the passed widget its purpose; `JButton` responds `"ButtonUI"`. `UIDefaults` uses its stored information to find out the appropriate `"ButtonUI"` UI delegate class for the current PLAF. It then uses Java's reflection interface to instantiate the UI delegate and returns the delegate to `UIManager`, which passes it to `JButton`.

[3]Note that there is a difference between subclassing and purpose. While `MotifButtonUI` extends `ButtonUI` and `MotifRadioButtonUI` extends `MotifButtonUI`, `MotifRadioButtonUI` does not satisfy the `"ButtonUI"` purpose.

`JButton` then begins the process of installing the button UI delegate object. `JButton` first calls an internal method to uninstall the current UI delegate object (not shown in the diagram) and then calls `installUI(JComponent)` on the button UI delegate object, passing itself as the argument. The button UI delegate installs various default properties onto the button, some of which are determined by `UIManager` and others which are determined by `LookAndFeel`. At the same time, the button UI delegate object creates a PLAF-specific button event handler and installs it on the button object.

2.2.3 The Problems

Is this a bad design? Certainly, as the arcs in Figure 1 show, there is a high degree of coupling between the components to support the UI delegate installation process. However, these interactions are not the result of bad design: the design uses many advanced object-oriented concepts and is reasonable given the constraints of the the mechanisms available within Java.

Even if it is state-of-the-art, the design is not satisfactory. Many components contain unnecessary knowledge.

`JButton`, for instance, contains EEK because it has to worry about the PLAF architecture during the UI delegate installation process. `JButton` should not need to ask `UIManager` for an appropriate UI delegate instance, and it should not need to know about its `uiClassID`. `JButton` contains or inherits five methods with the sole purpose of supporting this process: `getUIClassID()`, `updateUI()`, `getUI()`, `setUI(ButtonUI)`, and `setUI(ComponentUI)`. If these methods were not present, `JButton` would be conceptually cleaner, permitting it to be modified with less risk of breaking the system, and permitting it to be reused without having to reuse the ability to change look-and-feels. In addition, it seems unnecessary for `BasicButtonUI` to worry about installing a PLAF-specific event handler on `JButton`. The emphasized arcs within Figure 1 are ones we would like to break by replacing the UI delegate installation process.

Given this EEK, constructing new widget classes is also a non-trivial task because of the complexity required to support the library-level architectural concerns. A new class would need to implement, inherit, or override many similar methods, the subtle nuances of which quickly become lost.

To reduce the EEK in Swing components, a better mechanism is needed to simplify them, thereby making them more maintainable, reusable, and extensible. We believe that implicit context is such a mechanism.

3 Implicit Context

Implicit context consists of a means, which we refer to as contextual dispatch, for altering and rerouting messages based upon the history of calls made within a system. In this section, we describe the concepts behind a mechanism to support implicit context, we demonstrate how to overcome the problems within the abstract examples from Section 2, and we discuss the use of a proof-of-concept implementation of implicit context to address the EEK identified earlier in the Java Swing library.

3.1 Contextual Dispatch

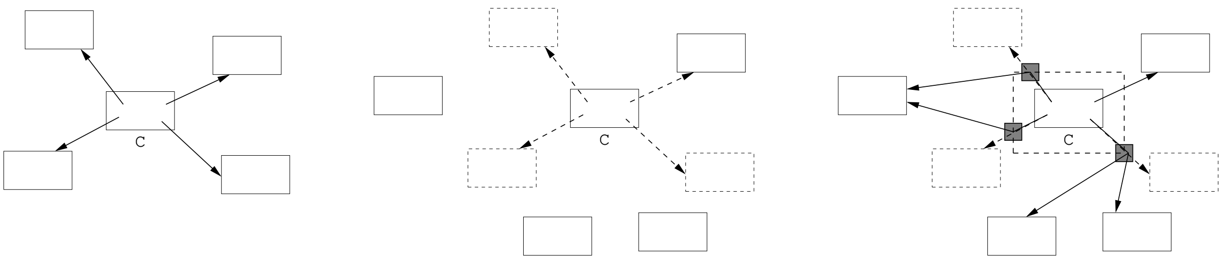
We want to be able to intercept messages before or after they are sent and to replace them by other messages depending on the context in which the calls occur. To do so requires a means of interception and a means of defining replacement messages.

Since the entire point of leveraging implicit context is to bring the implementation of a component closer to its conceptual requirements, it makes no sense to embed the interception and redispatch of messages within the component itself. Thus, we place the replacement method invocations in segments of code outside of the components they are to act upon; we call these segments *boundary maps*. There are two kinds of boundary maps: *out-maps* and *in-maps*.

Out-maps reroute calls from a component. Consider a component `C`, which through various method calls, names four external components within its system, `S1` (see Figure 2a). Although `C` insists that these external components be present within its system, we want to use `C` in a new system, `S2`; system `S2` contains different, but functionally similar, external components than `S1` (see Figure 2b). To match the requirements of `C` with the actual external components of `S2`, we consider there to exist a boundary between `C` and the other components in `S2`. When a message crosses this boundary it is intercepted and redispatched contextually in accordance with the out-maps (see Figure 2c).

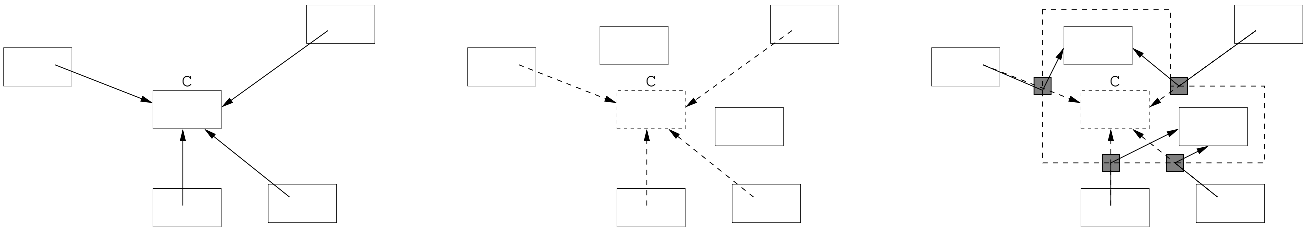
In-maps reroute calls entering a component. Consider a system in which a component, called `C`, is called by various other components (see Figure 3a). Although the other components depend on the presence of `C`, it is to be replaced by two components (see Figure 3b) with similar net functionality. To overcome the disparity between the expected and actual components, we construct a boundary around `C` and insert the new components within this boundary. In-maps placed on this boundary intercept messages from the rest of the system on their way to `C` and reroute them appropriately to the new components (see Figure 3c).

Boundary maps maintain the façade of an unchanging interface, thereby permitting a simple means of backwards compatibility. Out-maps help an individual component to possess an unchanging view of the system in which it runs, while in-maps can help a system to possess an unchanging view of individual components within it even when they are replaced or modified.



(a) *System `S1`* (b) *System `S2`* (c) *Using out-maps to redirect calls from `C`*

Figure 2: *Using out-maps to move a component to a new system.*



(a) *System with component `C`* (b) *System with two replacements for `C`* (c) *Using in-maps to redirect calls to `C`*

Figure 3: *Using in-maps to replace a component with two others.*

3.2 Call History

In order to reflect upon the history of calls made within a system, we need both a means to record the calls made within that system, and a means to access this record. The kind of queries used to access the call history largely determine the form of the information that must be recorded.

To locate calls of a particular form, there are several selection criteria that should be available: the class, method, and object receiving the call, and the parameters passed within the call. It also must be possible to determine the relative order of calls and to determine the causal relationships between calls. For instance, we must be able to support a query of the form: 'Was method `mA` in the call stack when method `mC` was called?"

3.3 Simple Examples

Recall the extraneous parameter example of Section 2. We can use boundary maps to remove knowledge of the extraneous parameter `param` from method `mB`. Optimally, `mB` would never have contained knowledge of `param` since it never uses it. We can accomplish this with an out-map attached to `mA`, which implicitly stores `param` within the call history, and an in-map attached to `mC`, which retrieves `param`. This would give `mC` access to `param` while allowing `mD` to ignore it. Method `mB` can now be replaced or altered, leaving the boundary maps in place to maintain the protocol of passing `param` or not.

Likewise, the protocol support example of Section 2 becomes less problematic with contextual dispatch. Given the ability to query the call history for whether `init()` has been called previously, we can remove all vestiges of support for the initialization protocol of class `Cls` from both the `doit()` method and all of the clients of `Cls`. Instead, an in-map is applied to `doit()` that checks for a call to `init()`; if such a call has not happened, `init()` is called prior to control passing to `doit()`.

3.4 Application of Implicit Context to Swing

To reduce the EEK identified earlier in Swing, we applied implicit context. We had three specific goals in mind:

1. remove the need to explicitly install PLAF-specific UI delegates onto `JButton`, thereby removing all details of the uninstallation/installation process from `JButton`,

2. remove the need to explicitly install PLAF-specific event handlers onto `JButton`, and

3. meet goals 1 and 2 in such a way that the rest of Swing continues to operate using the original PLAF architecture.

We chose to remove the PLAF architecture only from `JButton` for two reasons: we would want any implicit context approach to be incrementally adoptable, and Swing is too large and complex to tackle at once for a proof-of-concept.

There were five steps involved in applying implicit context: implement call history, remove the details of the PLAF architecture from `JButton`, determine the appropriate boundary maps to apply, apply the boundary maps, and verify that an application runs with the two architectures (PLAF-explicit and PLAF-implicit context) operating together.

3.4.1 Implementing Call History

Our implementation of call history for Java stores method calls and method returns within a threaded tree structure. Each node within the tree represents a call to a method within the program, including the receiving object, objects and primitives passed in the parameters, an object representing the class being called, and an object representing the method being called. Each of these nodes is an object of the class `Call`. Every `Call` node has a link to an associated `CallReturn` object in which the return value of that call is stored. The thread within the tree records the causal order of method calls.[4]

This tree is encapsulated within a class called `CONTEXT`.[5] `CONTEXT` defines a number of methods for performing queries on the call history. Table 1 contains the ones that have been defined so far; these are not an exhaustive set of all queries that would ever be needed.

To store a call in the tree, we defined two snippets of code to instrument the methods in the system, one that was to be executed at the start of each method and one that was to be executed at the end of each method. These were inserted at the start and end of each method via AspectJ™, Xerox PARC's aspect-oriented programming [11] tool for Java.

Using this approach, we instrumented all methods within classes in the packages `javax.swing`, `com.sun.java.swing.plaf.motif`, and their subpackages. A total of 1,433 classes were instrumented.

3.4.2 Removing the PLAF Architecture from `JButton`

To meet our first goal of removing the PLAF uninstallation/installation protocol from `JButton` we removed the five methods providing this functionality from the class: `getUIClassID()`, `updateUI()`, `getUI()`, `setUI(ButtonUI)`, and `setUI(ComponentUI)`. `JButton` is now free of the EEK arising from the PLAF uninstallation/installation process.

- `getCallReturn(Call)`
- `precedes(Call, Call)`
- `hasBeenCalled(Class, Method, Object)`
- `findLastCallTo(Class, Method)`
- `findLastCallToFrom(Class, Method, Object, Object)`
- `findLastCallToAnySubclass(Call, Class, Method)`
- `findLastCallToAnySubclassFrom(Class, Method, Object)`
- `findLastCallToPassingSubclassOf(Class, Method, Class)`

Table 1: *The query methods defined on the* `CONTEXT` *class.*

3.4.3 Determining Appropriate Boundary Maps

To make `JButton` utilize implicit context in place of the PLAF architecture, and to patch up the holes we tore in interfaces and protocols by removing the PLAF architectural methods from `JButton`, we then needed to apply boundary maps to several classes.

First, we applied an in-map to `JButton` to intercept messages bound for the now removed `getUI()` operation. This in-map performed a set of call history queries to determine the current UI delegate for the button (pseudocode appears in Table 2). The key idea in this in-map is to determine whether any UI delegate with the `"ButtonUI"` purpose has been activated since the last time the button was painted, indicating that the UI delegate for `JButton` needs to be changed. Determining whether any UI delegate with the `"ButtonUI"` purpose has been activated required the application of an out-map to `UIDefaults` that informed `UIManager` of each individual UI delegate class associated with a PLAF when that PLAF was selected to be current.

To replace the need to explicitly install PLAF-specific event handlers on `JButton` instances, we introduced a generic `DefaultButtonListener` event handler class. This class consisted of empty methods for handling events. An in-map was applied to the `getListener()` method of `DefaultButtonListener` that determined the current UI delegate, and hence, the appropriate look-and-feel-specific event handler class, such as `BasicButtonListener`; events were then rerouted to an instance of this class. The need to explicitly install PLAF-specific event handlers, and the EEK this introduced, were gone.

A variety of other simple in-maps and out-maps were also

[4] We currently ignore the issue of method calls occurring in separate threads; all calls are collected in a single tree.

[5] `CONTEXT` is all capitalized simply to indicate that it should not be treated like any other class. Invoking its methods does not store anything to the call history, for example.

(1) Set `paintCall` to be the most recent call to paint this `JButton`.
(2) Set `assocCall` to be the most recent call to associate a UI delegate class with a PLAF.
(3) Set `uiClass` to `null`.
(4) Repeat (5)–(8):
(5) If `paintCall` is not `null` and is more recent than `assocCall`, just return the cached UI delegate object.
(6) Retrieve the UI delegate class passed in the `assocCall`.
(7) Set `uiClass` to the UI delegate's purpose.
(8) Set `assocCall` to be the next most recent call to associate a UI delegate class with a PLAF.
(9) : until the purpose is `"ButtonUI"`.
(10) Instantiate the UI delegate class and cache the object.
(11) Return the cached object.

Table 2: *Pseudocode for the `getUI()` in-map.*

needed to complete the integration of implicit context. An out-map was attached to `JButton` to reroute accesses on an internal UI delegate field to the `getUI()` in-map. Empty in-maps were attached to `JButton` for each of the PLAF architectural methods that we had earlier torn out, except `getUI()`, in order to maintain `JButton`'s interface. Finally, in-maps and out-maps were applied to the button UI delegate classes for a few initialization calls that had been made during the UI delegate installation process.

In all, `JButton` required 5 in-maps and 3 out-maps, `DefaultButtonListener` required 11 in-maps (for all the different event handler methods), `UIManager` and `UIDefaults` each required one in-map, `BasicButtonUI` required 8 in-maps and 5 out-maps, and each PLAF-specific UI delegate class (i.e., `MetalButtonUI` and `MotifButtonUI`) required 4 in-maps and 5 out-maps.

Statements to perform queries on the call history were used five times for `JButton` within the `getUI()` in-map, twice for `DefaultButtonListener` within the `getListener()` in-map, three times for `BasicButtonUI` within three in-maps, once for `MetalButtonUI`, and once for `MotifButtonUI`. All boundary maps except the in-maps for `getUI()` and `getListener()` were short: six lines of code or less. The in-maps for `getUI()` and `getListener()` are 25 lines of code each; most of this code resulted from handling the initialization case where the button has not been painted yet.

Since this was a proof-of-concept, we only altered the Motif and Metal PLAFs. Altering the other PLAFs would be straightforward.

3.4.4 Applying the Boundary Maps

To apply the boundary maps to components, we manually wove in the necessary redispatching code to the components. Attaching an in-map to a method simply required that the body of the map be inserted (i.e., cut-and-pasted) into that method prior to any statements in the original method, including statements to store into call history. In-mapping a method that did not exist within a class involved adding a method of the indicated name and signature with the specified body to the class.

Out-mapping a call or field access required that a new method be added to the class to which the mapping was applied; the new method contained the body of the out-map. Then, all call sites affected by the map were modified to call the new out-map method. Doing this manually simply required a lexical search to ensure that each resulting match was in the correct scope, followed by a replacement by the name of the call to the out-map method.

3.4.5 Verifying that It Works

We tested the resulting combination of the PLAF architecture with the replacement architecture for `JButton` based on implicit context by building a simple application consisting of a couple of buttons and labels. Pressing one of the buttons caused the PLAF to be toggled between the Motif PLAF and the Java cross-platform ("Metal") PLAF.

The behaviour of the implicit context-based architecture when the PLAF is changed is depicted in Figure 4. No arcs remain from `JButton` to `BasicButtonUI` or vice versa, indicating the removal of the installation process from `JButton` and the removal of the installation of a `BasicButtonListener` on `JButton`. Also, the protocol is now more centralized and separated from the concerns of the individual classes, as indicated by the clustering of the call sites within the in-maps.

3.4.6 Results of Applying Implicit Context

Applying implicit context to Swing had four effects on the Swing library:

1. the source code for `JButton` is now conceptually simpler and contains less EEK: the code focuses on implementing the functionality of a button;
2. `JButton` should be easier to reuse without needing to reuse the PLAF architecture;
3. `JButton` should be easier to maintain and evolve now that it is free of the concerns of the PLAF architecture; and
4. the components of the PLAF architecture should be easier to maintain and evolve since they are now free of EEK related to the core concerns of `JButton`.

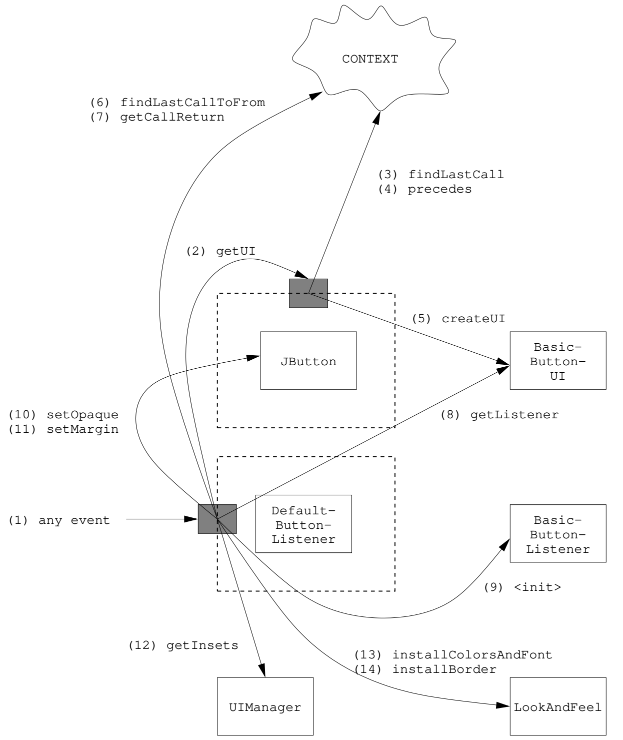


Figure 4: *The behaviour of the implicit context-based architecture when the PLAF is changed. The shaded boxes represent the in-maps attached to* `JButton` *and* `DefaultButtonListener`.

4 Discussion

Despite the advantages gained applying implicit context to Swing, given the early stage of work on implicit context, many open issues remain.

4.1 Tool Support

To date, we have applied implicit context with minimal automated support. Making the concept of implicit context workable obviously requires tool support for both contextual dispatch and for recording and querying call history. Currently, we are developing tool support for applying implicit context to Java programs.

4.1.1 Contextual Dispatch

As described earlier, applying in-maps and out-maps to Swing components was a straightforward process, involving the addition of new methods and modification of existing methods in classes with associated maps. Using this approach, components (classes) can be processed individually. Building a tool to perform this process requires support for processing map specifications and manipulating Java source code. Conceptually, the tool support needed is straightforward. However, since we are working with an object-oriented language such as Java, we must deal with such issues as inheritance and protection levels. The details regarding these issues are beyond the scope of this paper.

4.1.2 Call History

Gathering of call information can be achieved by instrumenting each method of each class to record, for every invocation, the necessary information in the history. Our experience in instrumenting systems to support object-oriented visualization [18] suggests that this approach will have a significant (negative) impact on performance. Luckily, the in-map and out-map specifications can help. These specifications can be analyzed to determine the subset of methods that must be instrumented, reducing the amount of call information that needs to be recorded. Although this approach will require a global analysis of the components and maps that are to be used together, we believe this is workable for two reasons. First, the analysis is not heavyweight, requiring only a scan of the maps and of the static inheritance structure of the system. Second, the instrumentation that must be applied to gather the information requires only a simple transformation to the source code and can even be done at load time.

Collecting the call history information is just one part of the problem. Support must also be available to query this information efficiently. If efficient querying remains a problem after reducing the amount of information collected, we intend to build upon encoding techniques we have recently developed to support the visualization of large object-oriented systems.[6]

4.2 Effect on Structure

One potential criticism of implicit context is that EEK is not removed, rather it is simply moved to boundary maps. When an in- or out-map is used to redirect messages, without use of the call history, EEK is indeed moved and not removed. However, the movement of EEK is an advantage: in- and out-maps are intended to be simple and more modifiable than having to wade through and change multiple parts of a component. As an example, consider the implicit context version of our Swing example. Event though the in-map for `JButton` still interacts with `BasicButtonUI` as part of the installation of the PLAF, moving this interaction to the map makes it easier to understand the installation process and simplifies the code for `JButton` itself.

When the in-map or out-map uses the call history to determine the appropriate dispatch, EEK is indeed removed, reducing the dependences of one part of a system upon another part. In our modified version of Swing, for example, `JButton` no longer requires knowledge of look-and-feel purpose (i.e., the `"ButtonUI"` string).

[6] This work is implemented but not yet published. A technical report will be forthcoming early in the new year.

4.3 Effect on Performance

Our implicit context version of Swing is operational, but is not fast. Not surprisingly, the main cause of a degradation in performance compared to the unmodified Swing is attributable to the searches through the call history. We believe a combination of reducing the size of the history and optimized searching mechanisms can help address this problem.

4.4 Effect on Development

Another criticism of implicit context is that it makes it more difficult to reason about the operation of the system. For Swing, we believe our implicit context version is easier to reason about because it separates and simplifies a particular complex feature from the regular operation of `JButton`. Although several maps must be investigated to understand the feature, each map is relatively small; the largest is about 25 lines. Certainly, if more features of Swing were separated into maps, the maps may become overly complex themselves. This criticism also applies to other techniques that support separation of concerns (e.g., [16, 11]). More experience must be gathered applying these approaches to assess the impact.

5 Related Work

Much of the work in software engineering and programming languages is oriented at increasing the independence and reuse of components.

5.1 Structural Approaches

Implicit invocation (a.k.a. publish-subscribe, event multicast) [8] 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 (i.e., the callback registrar, subscribers, and event publisher) need to be aware that this particular mechanism is in place, and subscribers and event publishers need to recognize a common interface for passing events and what those events are. Implicit context allows protocol concerns to be separated and allows protocol information to be adapted as necessary. For instance, parameters expected by a subscriber might be modified as part of a boundary map.

DeLine's flexible packaging [5] is a mechanism by which decisions about how a component is to interact with others can be delayed until system integration time. Flexible packaging separates a component's functionality and its packaging into distinct entities: a *ware* and a *packager*. A given ware can then be packaged to work in different environments; for instance, as a web browser plug-in or as a command-line filter. DeLine's approach provides a more abstracted means of addressing the question of *how* a component interacts than the support provided by implicit context. This additional abstraction comes at a price: wares must be built to a particular abstracted notion of interaction. In contrast, implicit context can be used to adapt a component to work in a new situation without the component necessarily being aware of the adaptation.

A number of more general approaches to separating concerns in a system have been appearing over the last few years. Subject-oriented composition [16] is a means for composing and integrating disparate class hierarchies (subjects), each of which might represent different concerns. Aspect-oriented programming [11] provides support for modularizing crosscutting concerns, such as distribution or look-and-feel, in a system. Modularized concerns can then be woven in to a system as desired. Similar to these approaches, implicit context is intended to help separate different parts of a system, increasing the independence of those parts. In contrast, implicit context provides later binding of the pieces of a system to each other through access to the call history.

5.2 Explicit Context

Other approaches focus on the use of explicit context to increase the flexibility of a system.

Context relations [17] provide a language-based mechanism in support of the Strategy pattern [7] by allowing "context objects" to be dynamically attached to instances. Context reflection [15] allows interpretation of messages and knowledge in terms of an explicitly-set, globally current context, allowing late binding. Behaviourally adaptive objects [13] 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. All three of these mechanisms permit significant dynamic flexibility, and hence might address the need for eliminating early binding of names, but they do not provide any special means for coping with other forms of coupling such as extraneous embedded knowledge concerning protocol adherence or extraneous parameters.

Traces [10] 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 prespecified 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 separating the Swing PLAF architecture from `JButton` described earlier.

5.3 Remapping Approaches

Many object-oriented reflective systems support the reification and manipulation of messages or methods (e.g., [14, 11]). Most of this work does not discuss how the message manipulation should be structured to support software engineering goals. One approach that supports the remapping of messages that appear at a boundary in a structured way is composition filters [1]. In the composition filters model, an object consists of an internal part, possibly consisting of multiple objects, and an interface part, which defines input and output filters to manipulate and possibly redirect messages. In this model, filters are specified explicitly within classes, limiting the separation of EEK and limiting the reuse of classes. In addition, since composition filters do not provide access to the call history, filters cannot be used to address such problems as extraneous parameters.

Several other approaches provide programming language-based approaches to delay the binding of operations to names.

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). Subjectivity [9, 12] allows different method implementations to be executed for a message depending on the run-time type of the sender of the message. Both of these approaches increase the flexibility of a system, but they do not eliminate some forms of EEK that implicit context is able to eliminate, such as extraneous parameters.

5.4 Call History

Both LambdaMOO [4] and Perl [19] permit access to the current call stack, but to no other, prior calls. Unlike implicit context, neither provides a means for retrieval of passed parameters.[7] In addition, neither approach provides any means for separating out the manipulation of messages.

6 Conclusion

Real components are often complex. The complexity within a component rarely stems from one cause. Rather a component, over time, ends up with knowledge of other components that is not conceptually required for the component to provide its behaviour, yet is to difficult to remove. We refer to this unnecessary information as extraneous embedded knowledge (EEK). EEK occurs in many forms in components, including a reliance on particular names, extraneous parameters, and a need to adhere to implicit protocols.

In this paper, we have introduced the concept of implicit context as a way of reducing EEK in components. Implicit context combines a means for rerouting messages in a system—contextual dispatch—with an ability to reflect over the history of calls that have been made in a system. We have shown how implicit context helped reduce the EEK involved in the process of installing a new look-and-feel on a component from the Java Swing library.

Our work to date has focused on showing the utility of the implicit context approach. Given the benefits achieved in applying implicit context to Swing, our next step is to automate support for implicit context and continue to investigate its impact on program structure.

[7] In Perl, access to passed parameters is available in one special case, when this access occurs within the DB package, as described by Wall et al. [19, p. 148].

Acknowledgements

We thank Martin Robillard for providing insightful comments on an earlier draft of this paper. This work was funded by the Natural Sciences and Engineering Research Council (NSERC) of Canada. Java is a trademark of Sun Microsystems. AspectJ is a trademark of Xerox Corporation.

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] Robert DeLine. Avoiding packaging mismatch with flexible packaging. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 97–

106. ACM Press, 1999. (ICSE-21; Los Angeles, USA; 16–22 May).

[6] Amy Fowler. A Swing architecture overview: The inside story on JFC component design. `http://java.sun.com/products/jfc/tsc/archive/what_is_arch/swing-arch/swing-arch.html`, 23 August 1999.

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

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

[9] 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.

[10] 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).

[11] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997. (11th European Conference on Object-Oriented Programming; Jyväskylä, Finland; 9–13 June).

[12] 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).

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

[14] Jeff McAffer. Meta-level programming with CodA. In W. Olthoff, editor, *ECOOP '95—Object-Oriented Programming*, volume 952 of *Lecture Notes in Computer Science*, pages 190–214. Springer-Verlag, 1995. (9th European Conference on Object-Oriented Programming; Åarhus, Denmark; 7–11 August).

[15] 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).

[16] 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.

[17] 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.

[18] Robert J. Walker, Gail C. Murphy, Bjorn Freeman-Benson, Darin Wright, Darin Swanson, and Jeremy Isaak. Visualizing dynamic software system information through high-level models. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 271–283, 1998. (OOPSLA '98; Vancouver, Canada; 18–22 October). Published as *ACM SIGPLAN Notices* 33(10), October 1998.

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