Implicit Context: Easing Software Evolution and Reuse

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

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

ABSTRACT

Software systems should consist of simple, conceptually clean software 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 into a system in many forms, including dependences upon particular names and the passing of extraneous parameters. This paper proposes the use of implicit context as a means for reducing EEK in systems by combining 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.

Categories and Subject Descriptors

D.1 [**Software**]: Programming Techniques; D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; D.2.11 [**Software Engineering**]: Software Architectures—*information hiding*; D.2.13 [**Software Engineering**]: Reusable Software.

General Terms

Algorithms, Languages.

Keywords

Structure, flexibility, extraneous embedded knowledge, EEK, implicit context, call history, contextual dispatch.

1. INTRODUCTION

When we begin building a software system, we typically strive to design software 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.[1]

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 design pattern [4] 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. 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). In Section 2, we expand upon our description of EEK and give some simple examples.

To remove EEK from components, we propose using *implicit context*—knowledge of the components that exist in a system and of the messages that have been previously communicated between them—to interpret and to alter messages. When a message is passed or received, additional details (such as parameters) can be filled-in by reflecting upon what has previously occurred within the system, the *call history*. Furthermore, a message can be altered depending on to where it is being sent or from where 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*. In Section 3, we expand upon this description of implicit context and show how it may be applied to remove the EEK from the simple examples of Section 2. We then describe, in Section 4, a semi-automated proof-of-concept mechanism for using implicit context that we have used, and discuss issues that arise in providing fully automated support for the approach.

To demonstrate the approach, we present the application of our proof-of-concept implementation of implicit context to part of the 1,304-class Java™ Swing graphical user interface library in Section 5. 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.

In Section 6, we discuss issues that arise in using implicit context. Section 7 compares implicit context to other, related approaches. Finally, we summarize our arguments and findings in Section 8.

[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.

SIGSOFT 2000 (FSE-8) 11/00 San Diego, CA, USA
 ISBN 1-58113-205-0/00/0011...$5.00

The contributions made by this paper are: (1) to illustrate the problem of extraneous embedded knowledge, and (2) to introduce implicit context, a concept whose use can reduce EEK, thereby easing software evolution and reuse.

2. EXTRANEOUS EMBEDDED KNOWLEDGE

Extraneous embedded knowledge creeps into a component as the component is elaborated and implemented. Sometimes the dependences we build in can come back to haunt us later when we need to change our system or reuse pieces of it. We focus on a few of the more common forms of EEK here.[2]

The simplest form of EEK consists of the dependences a client component forms on particular names and signatures of external components. If any of the external names or signatures change, our client component will break. What should be important to the client is not *which* component will be providing desired external functionality, but rather *what* functionality is needed. For example, say we had a client that referred to the `Vector` class; at some point `Vector` is renamed `DynamicArray`, thereby breaking our client. Our client cared about the `Vector` functionality, not the name; therefore, the specific name was EEK within the client.

More complex EEK arises, for example, when a class participates in the Abstract Factory design pattern [4] as a `Client`. `Client` must explicitly name the `AbstractFactory` class and alter the way in which it would otherwise create an instance of an `AbstractProduct` class. Rather than containing statements of the form (using Java syntax):

```
AbstractProduct product =
    new AbstractProduct();
```

`Client` must use the more convoluted form:

```
AbstractProduct product =
    factory.makeAbstractProduct();
```

where `factory` is of type `AbstractFactory` and contains an instance of `ConcreteFactory`, a subclass of `AbstractFactory`, that has been passed to the `Client` at some point. Therefore, `Client` contains explicit knowledge of its participation in the Abstract Factory design pattern: it both names the `AbstractFactory` class and invokes a factory method therein. The presence of this knowledge within `Client` forces the Abstract Factory design pattern to be used whenever `Client` is to be reused. Since `Client` would be just as effective at providing its intended purpose were it possible to use the first form of the statement above, the knowledge of the Abstract Factory design pattern involved in the use of the second form is EEK within `Client`.

A more subtle version of EEK arises between components. Consider drawing toy robots represented by a `Robot` class. `Robot` possesses a `draw` method that delegates to a series of other methods for drawing the different parts of the robot: the torso, head, left arm, and so forth. Each of these methods, in turn, delegates to yet other methods for drawing smaller bits of the robot, primitive objects (such as rectangles), or both. In implementing these methods, we realize that we need a canvas on which to draw primitives. Because the robot may need to be drawn on different canvasses, a canvas object must be passed to the `Robot` class; therefore, a parameter is added to the `draw` method to accept such an object. Since we need to have this canvas at the points where it is required to draw primitives, a parameter is added to all the methods that draw parts of the robot and each method dutifully passes the canvas on. Only the methods that draw primitives actually do anything with the canvas; the others simply pass it to the methods to which each delegates. Knowledge of the canvas can thus be seen to be EEK from the perspective of these methods: it is needed not for the conceptual integrity of each of these methods but only because a canvas object needs to be explicitly used to draw primitives, and this object must travel from the clients of `Robot`, through the hierarchy of delegations, down to the primitive drawing calls being made.

Categorization of knowledge as EEK can only occur relative to a particular component. For example, the fact that knowledge of the Abstract Factory design pattern is EEK within `Client` does not imply that it is EEK within a larger, parent component containing `Client`. This parent may indeed be very concerned with the fact that the Abstract Factory design pattern is being used in favour of some other means of flexibility. If this were the case, it would not make sense to talk of reusing the parent without the Abstract Factory design pattern and, since the pattern would be an integral part of the parent, it would not be EEK there. Likewise, primitive drawing methods need to refer to a canvas for their operation to be meaningful. Therefore, a canvas object is not EEK within such a primitive drawing method.

3. IMPLICIT CONTEXT

Implicit context is the context provided by the execution of a system: at any given moment during an execution, the implicit context consists of the structure of the system and the history of interactions within the system. We can interpret messages differently depending on the implicit context in which they are being communicated.

As an analogy, consider that in human conversation we do not spell out every concept we wish to communicate at every instant 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: *omission*, where words or details are left out to be filled-in from earlier details within a conversation or general knowledge, and *alteration*, 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 omitted, to be understood from what has previously been discussed. Likewise, one's response to the question, "What is politics?" might be quite different depending on who is asking; the explanation given a young child is likely to be significantly different from that given an adult. Meaningful use of context can require that the participants in a conversation share a common *world view* when referencing knowledge outside the confines of the conversation. Stating that someone "acted the role of Cyrano" would be meaningless if the listener knew nothing of Cyrano de Bergerac.[3]

Similarly, the history of messages within a system can be viewed as a conversation between components, and so, we can leverage the concepts of omission, alteration, and world view. Rather than forcing components to repeatedly give the same details in messages, we wish to allow them to send messages with omitted details, their meanings to be understood from implicit context. At the same time, we want to perform alteration of messages

[2] EEK is a generally applicable concept; however, we limit its discussion to object-orientated software within this paper due to a need to ground the discussion and due to space limitations.

[3] Our analogy is intended to be motivational. In human conversation, we also perform operations such as checking that we have understood what is being discussed. We are not attempting to provide such operations via implicit context.

depending on where they are received or to whom they are sent. And finally, components need to share a common world view, or apparently share one, so implicit context can be correctly used.

With these concepts in hand, we proceed to discuss a programmatic model for using implicit context, and demonstrate its utility on the simple examples from Section 2. We postpone a description of approaches to implementing a mechanism to use implicit context until Section 4 and a discussion of how existing approaches address EEK until Section 7.

3.1 The Model

In order that the messages which are sent and received be modifiable according to the implicit context, we need to be able to intercept messages, reflect upon the implicit context, and alter those messages accordingly.[4] Our model for utilizing implicit context separates the interception, alteration, and redispatch of messages, called contextual dispatch, from the reflection upon the call history. Ideally, contextual dispatch provides the mechanism for rationalizing disparate world views between intercommunicating components, interpreting messages with omitted details, and altering messages contextually. The call history and some structural information provide the knowledge with which to drive contextual dispatch.

In the model, we can consider there to exist a boundary around every component. Inside the boundary, the world view of the component holds sway; outside the boundary is either the true global picture, or the world view of a larger, nesting component. Translation between the external and internal world views occurs at the boundary within what are termed *boundary maps*. There are two kinds of boundary maps: *out-maps* and *in-maps*. Out-maps translate from the internal world view to the external world view, while in-maps translate in the opposite direction. Each boundary map is responsible for the interception of a particular kind of message from one side of the boundary and its contextual dispatch, generally to the other side of the boundary. We say that we *attach* or *apply* a boundary map to a boundary when the boundary map is explicitly associated with that boundary.

In Figure 1, we see a component `C` with a boundary shown around it; `C`, through various method calls, names four external components, `S1`–`S4`, within its world view. The external world view is different; it contains four components, `T1`–`T4`. There is only one component that matches between these world views: `S2` and `T2` are the same. The world view represented by the `S` components could be the system in which `C` was originally implemented and the `T` components could be a system in which `C` was reused. In order that `C` operate correctly, we define three out-maps for it (shown as grey circles) that intercept messages bound for the nonexistent `S1`, `S3`, and `S4`. Messages bound for `S1` and `S4` are always rerouted to `T1`. Messages bound for `S3` are rerouted to either `T3` or `T4` on the basis of the call history; it could be that the least recently used one is always used, or it might depend on some initialization choice that occurred previously. In-maps behave analogously.

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

[4] Just as we present our discussion of EEK from an object-oriented view, we present our discussion of implicit context from an object-oriented view. However, implicit context could be used in any situation where information flows across a recognizable boundary.

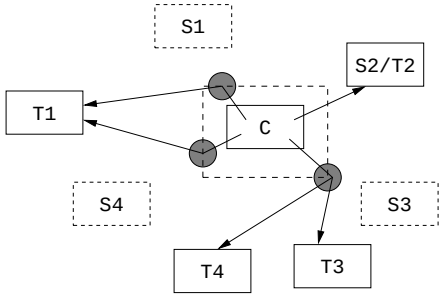

Figure 1: Using out-maps to redirect calls from `C`.

3.2 Removing EEK: Simple Examples

Dependences on names of external components are trivial to repair. Boundary maps capture messages bound for the purported components, and replace them with messages to the actual components. Thus, in our example, messages bound for `Vector` are replaced with messages bound to `DynamicArray`. The EEK imposed by requiring client components to know what other components exist in the real system has been removed from the clients.

Recall the Abstract Factory design pattern example of Section 2. To permit `Client` to literally contain the statement:

```
AbstractProduct product =
    new AbstractProduct();
```

while actually utilizing an abstract factory, we must capture the call to `AbstractProduct` in an out-map attached to the boundary of `Client`. In this out-map, we query call history to determine the last instance of `AbstractFactory`, which we will refer to as "`factory`," that was passed to `Client`. The message carrying `factory` was ignored through an in-map also attached to the boundary of `Client`. We then reroute the call to `AbstractProduct` to go to `factory.makeAbstractProduct` instead, and return the resulting object to `Client`. Note that, as a result of the boundary maps, `Client` does not require any actual constructor `AbstractProduct` to exist to receive this message, since the message is intercepted before it gets there.[5] `Client` and its boundary are both part of some parent component; by moving statements to the boundary, they remain part of the parent, but not part of `Client`. The EEK in our system has now been reduced: the Client no longer has any knowledge of the Abstract Factory design pattern, only the parent has this knowledge and this is not EEK as described in Section 2.

Our robot example is also straightforward to deal with via implicit context. Again, we attach an in-map to the boundary of `Robot` that filters out the canvas object that gets passed to it. We attach an out-map to the boundary that, for each primitive drawing call, first looks up the previously passed canvas and either adds it as a parameter to the drawing call, or reroutes the drawing call to the canvas itself. All references to the canvas have been removed from `Robot`; references to the canvas remain in the boundary maps to `Robot`, but, as described before (see Section 2), it is not EEK there.

[5] In our implementation described in Section 4, the references made by this code to the constructor for `AbstractProduct` are replaced prior to compilation with a call to `factory.makeAbstractProduct`, and therefore, the transformed `Client` does not require that the constructor exist.

4. IMPLEMENTATION

We have implemented a mechanism for recording call history and a means for performing contextual dispatch in Java. This proof-of-concept implementation was built for the purpose of determining whether the concepts envisaged in implicit context were worthy of further study; as a result, the implementation attempts to provide a literal representation of the model presented in Section 3. Not surprisingly, mismatches exist between the general model and what is possible within Java, so some details can not be perfectly realized; we have noted where we have made tradeoffs.

We begin with a description of the implemented means of recording and querying call history, then explain how boundary maps have been defined and applied, and end with a brief discussion of how we are addressing automated tool support.

4.1 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 kinds of queries used to access the call history largely determine the form of the information that must be recorded.

Our proof-of-concept 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, ignoring the presence of separate threads. This tree is encapsulated within a class called `Context`.[6]

A number of methods were implemented on `Context` for performing queries on the call history; Figure 2 contains a list of these. This is not an exhaustive list of all possible queries.

To store calls and call returns in the tree, we defined two snippets of code to instrument the methods in a 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. The source code for the classes was then so instrumented by automated tool support.

4.2 Contextual Dispatch

Boundary maps are the heart of contextual dispatch, but they require boundaries to which to apply. For our proof-of-concept, we considered only natural and easily named boundaries provided by Java, those around methods and fields.[7] There are two facets to boundary maps: specification of the messages that they should intercept and the redispatching code that should be executed when the corresponding messages are intercepted. The model calls for messages to be intercepted and modified, but in Java, there is no means to do this on actual messages; therefore, we statically modify methods to achieve the same effect on all method calls and outgoing field accesses.

We can see these two facets by examining a boundary map applicable to the Abstract Factory example from Section 2. The following out-map could have been applied to the boundary of `Client` to allow it to refer to `AbstractProduct` directly:

```
map abstractFactoryMap {
  out AbstractProduct() {
    AbstractFactory factory =
      Context.findLastInstanceOfPassedTo
                    (AbstractFactory.class,
                     Client.class);
    return factory.makeAbstractProduct();
  }
}
apply abstractFactoryMap to Client;
```

In this example, an out-map that intercepts messages to a constructor of `AbstractProduct` is applied to the boundary of `Client`. An identifier, `abstractFactoryMap` in the example, permits multiple in- and out-maps to be given a single name, making it easier to apply a set of maps to a component. The one out-map in `abstractFactoryMap` replaces messages to the `AbstractProduct` constructor with the indicated block of redispatching code. This block makes a call to `Context` to locate whichever instance of `AbstractFactory` was last passed to `Client`. A factory method is then invoked on the located object.

To apply the boundary maps to component boundaries, we manually added the relevant redispatching code into the classes named in `apply` statements. Attaching an in-map to a method boundary simply required that its redispatching code block 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. We did not permit field accesses to be in-mapped because there is no means of capturing accesses to the fields of a class in Java.[8]

Out-mapping a call or field access involved adding a new method to the class to whose boundary the mapping was applied; the new method contained the redispatching code block of the out-map. Then, all call sites affected by the map were modified to call the new out-map method. Doing this manually required a search to

- `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)`
- `findLastInstanceOfPassedTo(Class, Class)`

Figure 2: The query methods defined on the `Context` class.

[6] Note that the `Context` class is treated specially: invoking its methods stores nothing to the call history and it is intended to be accessed only within boundary maps.

[7] Classes are also natural and easily named in Java, but boundaries around them were approximated for now by applying the same boundary maps to every method and field therein.

[8] All the client classes who access a field could have had equivalent out-maps attached to their boundaries, but we wanted to maintain the concept of independent compilation of classes.

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. An additional parameter is required by the out-map method to hold the object to which the original message was to be sent.

Although we did not encounter the need in our example study, the model accounts for the application of multiple maps to a given boundary. These are applied sequentially in the order specified by the engineer, and each added map modifies the code added by those before it.

4.3 Automated 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 automated tool support for utilizing implicit context in Java programs.

As described earlier, applying in-maps and out-maps is 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 boundary map specifications and manipulating Java source code.[9] We are using JavaCC[10] from Metamata to parse Java source and boundary map code. We are manipulating the resulting tokens to achieve an effect identical to the manual insertion process described above.

5. EXAMPLE: 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 [3]. This architecture allows the display and interactive characteristics (the "look-and-feel") of a GUI to be altered dynamically; for example, a user interface in the Motif look-and-feel can be altered at *run-time* to a Windows look-and-feel and back again.

5.1 EEK in `JButton`

In Swing, each GUI widget object contains a separate object, called a *UI delegate*, which is responsible for the look-and-feel of the widget for a particular PLAF. For example, the `JButton` class, 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 currently 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.

A button is a conceptually simple thing, yet the `JButton` class defines or inherits a total of 183 public methods within the `javax.swing` package, plus 144 public methods from within the `java.awt` package. We consider much of this to be EEK from the perspective of the `JButton` class. For instance, knowledge about the PLAF architecture is EEK from the perspective of `JButton` because it should be possible to reuse button widgets in the absence of the PLAF architecture.

To identify how `JButton` (and ultimately, the rest of the widget classes in Swing) could be evolved to remove details of the PLAF architecture without breaking Swing, we first need to examine the details behind the operation of the PLAF architecture.

5.2 How PLAF Works

Figure 3 shows a simplified object interaction diagram for the process of locating, instantiating, and installing a new UI delegate into a `JButton` object. There are five classes involved in this process aside from `JButton`.

- `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.
- `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.

The interactions between these five classes to support the changing of the look-and-feel are complex. Figure 3 depicts most of 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 (1). For `JButton`, this results in a request (2) to `UIManager` to obtain a UI delegate object that is appropriate to the new PLAF. `UIManager` locates the current PLAF (3, 4) and passes it (5) with the widget asking to be updated to `UIDefaults`. `UIDefaults` asks the passed widget its purpose (6); `JButton` responds `"ButtonUI"`. `UIDefaults` uses its stored information (7) 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 (8) and returns the delegate to `UIManager`, which passes it to `JButton`.

`JButton` then begins the process of installing the button UI delegate object (9). `JButton` first calls an internal method to uninstall the current UI delegate object (not shown in the diagram) and then calls `installUI` (10) on the button UI delegate object, passing itself as the argument. The button UI delegate installs various default properties onto the button (15–17), some of which are determined by `UIManager` (11) and others which are determined by

[9] This manipulation could also be performed on class files.
[10] `http://www.metamata.com/JavaCC/`

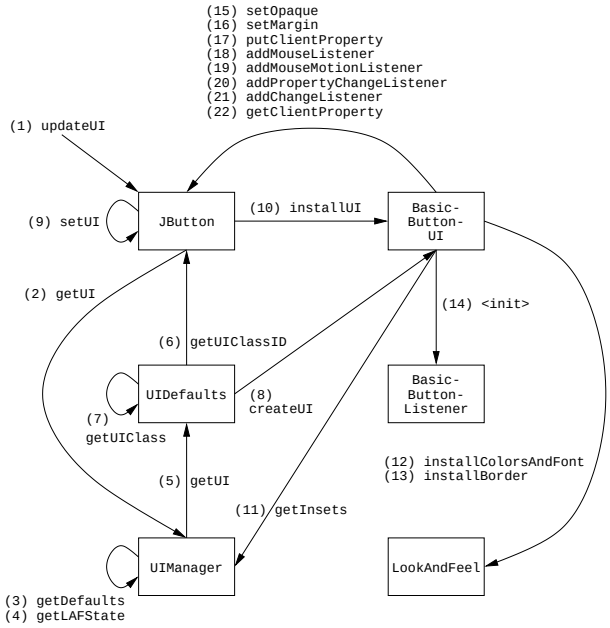

Figure 3: Object interaction graph for the process of installing a "Basic" PLAF UI delegate into a `JButton`.

`LookAndFeel` (12, 13). At the same time, the button UI delegate object creates a PLAF-specific button event handler (14) and installs it on the button object (18–22).

5.3 Applying Implicit Context to `JButton`

In part, `JButton` 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 is EEK for `BasicButtonUI` to worry about installing a PLAF-specific event handler on `JButton`.

We had three specific goals in mind in applying implicit context to `JButton`:

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

2. have the PLAF of `JButton` remain dynamically changeable, and

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

There were three steps involved in applying implicit context: remove the details of the PLAF architecture from `JButton`, determine boundary maps to support the goals, and apply the boundary maps (as described in Section 4).

5.3.1 *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)`. To maintain the same externally visible interface to `JButton`, in-maps were applied to its boundary that captured messages to each of these methods and ignored them. `JButton` was then free of the EEK arising from the PLAF uninstallation/installation process. Since this broke the PLAF architecture and thus Swing, we needed to use implicit context in place of the EEK.

5.3.2 *Determining Appropriate Boundary Maps*

To repair the damage to Swing produced by removing the PLAF architecture from `JButton`, we needed to apply boundary maps to several class boundaries. (The resulting architecture is shown in Figure 5.)

The in-map attached to `JButton`'s `getUI` method performs a set of call history queries. These 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. Pseudocode for the in-map appears in Figure 4.

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 attached to the boundary of the `getListener` method of `DefaultButtonListener` that determines the current UI delegate, and hence, the appropriate PLAF-specific event handler class; events are then rerouted to an instance of this class. The need to explicitly install PLAF-specific event handlers on `JButton` and the EEK this introduced are now gone.

A variety of other simple in-maps and out-maps were also used to complete the integration of implicit context. 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,[11] and once for each PLAF-specific button UI delegate class. 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.

5.3.3 *Results of Applying Implicit Context*

We tested the results of our changes by building a simple application whose PLAF was changed dynamically. The behaviour of the implicit context-based architecture when the PLAF is changed

[11] Only one of these uses is significant; the others are there to make sure that an error occurs if the old architecture is being used from a `JButton`.

```
(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)  If paintCall is not null and is more recent than
     assocCall, just return the currently cached UI del-
     egate object.
(5)  Retrieve the UI delegate class passed in the assoc-
     Call.
(6)  Set uiClass to the UI delegate's purpose.
(7)  Set assocCall to be the next most recent call to as-
     sociate a UI delegate class with a PLAF.
(8)  If uiClass is not "ButtonUI," go to (4).
(9)  Instantiate the UI delegate class and cache the object.
(10) Return the cached object.
```

Figure 4: Pseudocode for the `getUI` in-map.

is depicted in Figure 5. No arcs remain from `JButton` to `BasicButtonUI` or vice versa, and no arcs remain from `JButton` to `UIManager` or vice versa; this indicates the removal of the installation process from `JButton` and the removal of the installation of a `BasicButtonListener` on `JButton`.

Applying implicit context to Swing had three 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; and
3. `JButton` should be easier to maintain and evolve now that it is free of the concerns of the PLAF architecture.

6. DISCUSSION

Despite the advantages offered by applying implicit context, given the early stage of this work, many open issues remain.

6.1 Effect on Comprehensibility

Implicit context may make it more difficult to reason about the operation of a system. We discuss two possible criticisms here.

One criticism is that comprehensibility is decreased by the separation of small pieces of code into maps which affect component code. This criticism also applies to other techniques that support separation of concerns (e.g., [10, 8]). More experience must be gathered applying these approaches to assess the impact. An initial assessment of aspect-oriented programming [15] found that the number of file switches during certain programming tasks when using aspect-oriented programming were comparable to or slightly less than when using conventional object-oriented programming; this was interpreted as indicating that any confusion created by the division was compensated for by the greater clarity of the divided pieces. It would be wrong to ascribe too great a confidence in these weak results, but, to date, it is the only empirical evidence available.

Another criticism is that boundary maps could become complex and difficult to understand. For Swing, we believe our implicit

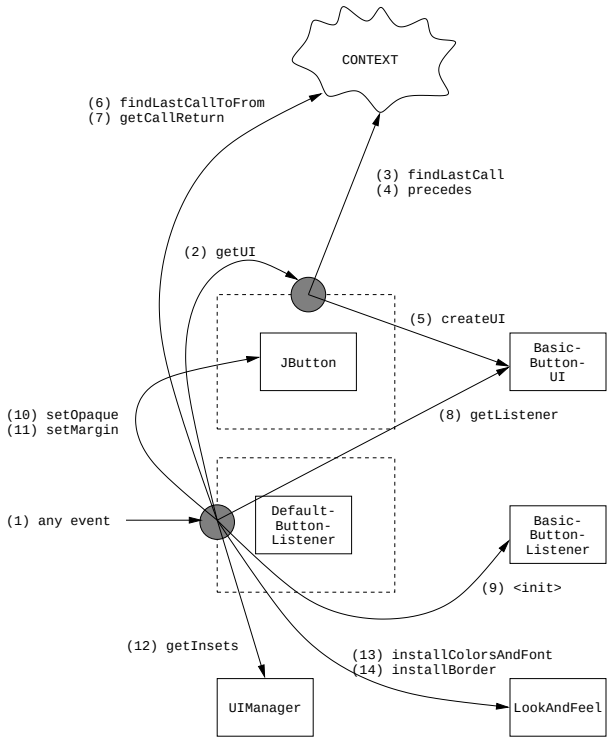

Figure 5: The behaviour of the implicit context-based architecture when the PLAF is changed. The shaded circles represent the in-maps attached to the boundaries of `JButton` and `DefaultButtonListener`.

context version is easier to reason about because it separates and simplifies a particular complex feature from the regular operation of `JButton`. Although several boundary maps must be investigated to understand the feature, each map is relatively small; the largest is about 25 lines. While the majority of in- and out-maps were trivial, the pseudocode given in Figure 4 for the `getUI` in-map is not trivial. Some of the complexity in this map is due to other existing intricacies of the PLAF architecture. Were we to more thoroughly apply implicit context to this example, the maps would likely become simpler. However, one of the advantages of the implicit context approach is that we are able to apply it either to a new system or incrementally to an existing system.

In our limited experience, once an understanding has been gained of the feature in a system to be modified, such as the PLAF architecture, determining which maps are needed and writing those maps is not an onerous task. Some maps are more complex than others. In these cases, the developer must reason through the system to determine how to access the information of interest in the past execution of the system. The difficulty of this task will vary depending on the complexity of the system and on the complexity of the EEK that is being removed. Further experience is needed to develop a better sense of the complexity or simplicity brought about by the use of implicit context.

6.2 Effect on Development

The use of implicit context allows us to remove EEK from systems. Implicit context does not merely move EEK around in a system since a determination of what is EEK is relative to a component. As described in Section 2, when EEK moves out of a component to the component's boundary, the code placed in the boundary is

generally not EEK to the higher-level, nesting component. As an example, consider the implicit context version of our Swing example. Even though the in-map for `JButton` still interacts with `BasicButtonUI` as part of the installation of the PLAF, moving this interaction to the boundary map separates the PLAF architecture from the code for `JButton` itself, thereby making it possible to reuse `JButton` without the PLAF architecture. The separation would ideally make it easier to understand and modify both `JButton` and the PLAF architecture in isolation although more experience is required before this can be assured. When the boundary map uses the call history to determine the appropriate dispatch, EEK is more obviously 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).

In realizing systems, flexibility mechanisms, such as certain design patterns, are currently chosen at some point before or during implementation according to the perceived needs of the system. We do not claim that these flexibility mechanisms are "bad" in and of themselves. It is the fact that these mechanisms tend to pervade components that is "bad"—they become EEK. If the wrong kind of flexibility has been chosen, altering our system to meet new flexibility demands is hard; altering components is hard because the flexibility mechanisms can obscure core concerns of components, and can even be broken when changing those core concerns. Implicit context allows us to gain the benefits of these flexibility mechanisms without being permanently tied to them and without the need to be concerned with them while developing or evolving components.

Although this paper has concentrated on the use of implicit context in changing an existing system, it also has applicability to the initial development of a system. If we could prevent the introduction of EEK during the inception of a system, the hope is that the system would simpler to create. Whether such speculation is more than wishful thinking is a matter for future work.

6.3 Potential Implementations

Our current implementation of a mechanism to use implicit context is simplistic; it is intended as a means of studying whether implicit context can increase the reusability and flexibility of components. The test application we used in our Swing example experienced an order of magnitude slowdown when using our proof-of-concept implementation. For implicit context to be useful, better implementations are needed; we argue here that such implementations are possible.

There are two obvious drawbacks to our proof-of-concept implementation. First, it stores call history by instrumenting each method of each class; for every invocation, the information related to the call is recorded in the history. Our experience in instrumenting systems to support object-oriented visualization [16] suggests that this approach will have a significant, negative impact on performance, and will require a great deal of storage for long-running programs. Second, it supports an arbitrary set of queries on call history. The more powerful that a query is, the more information that must be collected to support it.

These problems are addressable. The information that must be stored for some simple queries is optimizable; for example, the information required to answer the `precedes(A, B)` query could be stored as a single bit: the bit would be set when `B` is called and reset when `A` is called. Such optimizations can likely be found for a necessary subset of kinds of queries. Furthermore, we might only record the information that is actually needed to answer the queries made in a given system. 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 tools for the analysis of large object-oriented systems [17].

To perform these optimizations, 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 code and can even be done at load time.

We have not addressed these concerns yet since it is necessary to understand what form of the call history and queries are useful before we can consider optimizations.

7. RELATED WORK

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

7.1 Separation of Concerns

Implicit context is most closely related to those approaches intended to help explicitly separate concerns in program text. Some separation of concerns (SOC) approaches provide specific support for a particular kind of separation. Others provide a more general mechanism. We describe examples of each below and discuss how implicit context compares.

The relationship between EEK and concerns is unclear. While we believe that these are overlapping concepts, we also believe that the overlap is incomplete. EEK includes details too minor to be considered full-fledged concerns, such as name dependences, while a component could possess non-extraneous knowledge of multiple concerns. This relationship requires further investigation.

7.1.1 Specific SOC Approaches

DeLine's flexible packaging [2] focuses on separating the details about a component's interaction from the component itself. Flexible packaging separates a component's functionality and its interactions, called its packaging, into distinct entities: a *ware* and a *packager*. A given ware can be packaged to work in different environments, such as a plug-in for a web browser or a command-line filter.

Implicit parameters [9] allow one to explicitly designate when a set of intervening methods between a sender and a receiver do not need to be aware of a set of parameters: the parameters are transferred from the sender to the receiver without alteration of the source code for the intervening methods. The developer specifies within the source code for the sender and for the receiver that each parameter is to communicate via the implicit mechanism.

In comparison to implicit context, both flexible packaging and implicit parameters provide a more abstracted means of dealing with particular kinds of EEK. Flexible packaging provides a more abstract means of addressing the question of *how* a component interacts. Implicit parameters provide a specific means of expressing when a parameter is extraneous and of describing how to transmit it to the appropriate receiver. In each case, this additional abstraction comes at a price: the source for each component must be written to

explicitly use the particular mechanism. In contrast, implicit context can be used to achieve the same objectives without having to write each component to use the mechanism.

7.1.2 General SOC Approaches

A number of more general approaches to separating concerns in a system have been appearing over the last few years. Subject-oriented programming [6, 10] is a means for composing and integrating disparate class hierarchies (subjects), each of which might represent different concerns; subsequent work on hyperspaces [14] considers separating concerns in multiple dimensions at once. Aspect-oriented programming [8] provides support for modularizing cross-cutting concerns, such as distribution or look-and-feel, in a system. Modularized concerns can then be combined into a system as desired. Composition filters [1] separate objects into an internal part, possibly consisting of multiple objects, and an interface part, which defines input and output filters to manipulate and possibly redirect messages. Filters can be used to separate such concerns as synchronization.

Similar to these approaches, implicit context is intended to help separate different parts of a system, increasing the independence of those parts. Also similar to implicit context, each of these approaches involves explicit separation of parts of a program's source. Implicit context differs from these approaches in supporting contextual dispatch through reflection upon the call history of a system. This feature supports the investigation of how later binding of components to each other will affect the structuring of systems. Since all of these approaches are at an early stage of development, detailed analyses of the benefits and costs of each form of separation are not yet available.

7.2 Explicit Context

To increase the flexibility of a system, some approaches have focused on the use of *explicit* context.

Traces [7] allow the interpretation of messages to be altered based upon a limited form of explicit context. A list of "ancestor classes," which are independent of the class hierarchy, may be explicitly built and attached to an object. Also attached to each object is a set describing patterns of ancestor classes. When a message is received by an object, the choice of method to invoke is determined by which pattern the ancestor list of that object matches.

Context relations [13] provide a language-based mechanism in support of the Strategy pattern [4] by allowing "context objects," basically dispatch tables, to be dynamically attached to instances. Upon receipt of a message by an object, method selection is performed by the context object currently attached to the receiver object.

Both of these mechanisms help address a specific kind of EEK. The ancestor lists of traces can be thought of as particular paths through the call history tree, permitting a limited means of reflecting upon the system history. Context relations address the need for eliminating EEK related to the early binding of names.

The explicitness of these mechanisms forces a developer to commit to them at an early stage of development by intricately embedding their use within the source code, unlike with implicit context. Implicit context also permits the removal of more forms of EEK. For instance, in the implicit context approach, it is possible to access more information in the call history, such as the parameters related to a call. This additional information makes it possible to separate the Swing PLAF architecture from `JButton` described earlier. This kind of separation would not be possible using these explicit context approaches.

7.3 Adaptors and Wrappers

A variety of previous work attempts to decouple components either through alteration of interfaces, or by adding additional functionality behind an apparently unchanged interface.

Many of the structural design patterns [4] are attempts at this. For example, the Adapter design pattern alters the interface of a class so that it may be used by clients expecting a different one, while the Decorator design pattern allows additional responsibilities to be attached to an object dynamically. The Adapter design pattern has the disadvantage that, in its simplest form, it is not transparent to all clients since an adapted object no longer conforms to its original interface. Of course, if clients were to access the object only through the adapted interface, this would not be a problem. Otherwise, multi-way adaptation [4, p. 143] can be used to get around this problem, but it introduces the need for clients to instantiate a different class, i.e., the source code for the clients must be altered. Since the Decorator pattern works via delegation, there ensues the notorious problem of ensuring that the object calls its own methods via the decorator—the object identity problem.

POLYLITH [12] provides a "software bus" that allows the specifications of an application's structure, its deployment onto nodes, and inter-component communication to be separated. Simple communication statements within components are adapted to conform to the needs of the actual deployment geometry and communication protocol needs of a heterogeneous architectural- and language-environment. While POLYLITH is an attempt at making components more flexible by separating the concerns of distributed and inter-process communication, it cannot remove the same degree of EEK since it does not permit the indirect kind of communication that implicit context provides where, for example, parameters can be filled in from the call history.

Type adaptation [19] provides much the same mechanism as would contextual dispatch in the absence of call history. Call history allows us to fill in additional parameters in a way that type adaptation does not.

The goals of implicit context are similar to those of its adaptor and wrapper predecessors in that it allows a component to provide one interface while its clients expect another; as a result, components are reusable without invasive change being needed. However, implicit context is a mechanism that allows greater dissimilarity between the expected and provided interfaces. With implicit context, we are not constrained by the information being directly passed or directly accepted: we may add additional information garnered from the call history, or have information stored there for later use.

7.4 Similar Mechanisms

A number of existing approaches support similar mechanisms to those used in implicit context.

Implicit invocation [5] 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 names of subscribing classes and methods, but much remains: all components involved in an implicit invocation protocol relationship must be aware that this particular mechanism is in place, plus subscribers and event publishers need to recognize a common interface for passing events and what those events are.

A few languages (e.g., Perl [18] and Tcl [11]) permit access to the current call stack. Unlike implicit context, none of these provide access to prior calls. Nor does any of these provide a general means for the retrieval of passed parameters.

Dynamic scoping, available in languages such as some Lisp variants, also shares some similarity in mechanism to implicit con-

text. Dynamic scoping allows variable names to be bound into a non-lexical scope as determined by the call stack at run-time. In contrast to implicit context, dynamic scoping does not permit any description of what variable a name should be bound to, beyond that name itself; no properties of the scope in which the desired variable is to be found nor the structure of the call history upon which to base the search can be described. At the same time, we can learn from the experiences gained in using dynamic scoping as we elaborate the implicit context approach.

8. CONCLUSION

The reusability and evolvability of components are two goals of software engineering that are difficult to achieve with current technology. Using current approaches, components are often complex. The complexity within a component rarely stems from one cause. Rather a component will end up with knowledge of other components that is not conceptually required for the component to provide its behaviour, yet is 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, supporting non-local structure, and extraneous parameters.

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 with an ability to reflect over the history of calls that have been made in a system.

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 to continue to investigate its impact on program structure.

In a perfect world where we had software that never evolved and where we were not concerned about the cost of reinventing the wheel for every new system, implicit context would not be needed. Implicit context allows us to incrementally manipulate the structuring of systems to address the problems of reusability and evolvability encountered in software engineering.

9. ACKNOWLEDGEMENTS

We thank Siobhán Clarke and Martin Robillard for providing insightful comments on an earlier draft of this paper, Brian de Alwis for a discussion on flexibility and design, and Gregor Kiczales for a version of the canvas passing example. This work was funded by the Natural Sciences and Engineering Research Council (NSERC) of Canada. “Java” is a trademark of Sun Microsystems.

10. REFERENCES

[1] M. Akşit, L. Bergmans, and S. Vural. An object-oriented language–database integration model: The composition-filters approach. In *European Conference on Object-Oriented Programming*, pp. 372–395, 1992. LNCS 615.

[2] R. DeLine. Avoiding packaging mismatch with flexible packaging. In *International Conference on Software Engineering*, pp. 97–106, 1999.

[3] A. 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`, 1999.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[5] D. Garlan and D. Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *International Symposium of VDM Europe*, pp. 31–44, 1991. LNCS 551.

[6] W. Harrison and H. Ossher. Subject-oriented programming: A critique of pure objects. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 411–428, 1993. ACM SIGPLAN.

[7] G. Kiczales. Traces (a cut at the “make isn’t generic” problem). In *International Symposium on Object Technologies for Advanced Software*, pp. 27–43, 1993. LNCS 742.

[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, pp. 220–242, 1997. LNCS 1241.

[9] J. Lewis, M. Shields, E. Meijer, and J. Launchbury. Implicit parameters: Dynamic scoping with static types. In *Symposium on Principles of Programming Languages*, pp. 108–118, 2000.

[10] H. Ossher, M. Kaplan, A. Katz, W. Harrison, and V. Kruskal. Specifying subject-oriented composition. *Theory and Practice of Object Systems*, 2(3):179–202, 1996.

[11] J. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[12] J. Purtilo. The POLYLITH software bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151–174, 1994.

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

[14] P. Tarr, H. Ossher, W. Harrison, and S. Sutton. *N* degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering*, pp. 107–119, 1999.

[15] R. Walker, E. Baniassad, and G. Murphy. An initial assessment of aspect-oriented programming. In *International Conference on Software Engineering*, pp. 120–130, 1999.

[16] R. Walker, G. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak. Visualizing dynamic software system information through high-level models. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 271–283, 1998. ACM SIGPLAN.

[17] R. Walker, G. Murphy, J. Steinbok, and M. Robillard. Efficient mapping of software system traces to architectural views. Technical report TR-00-09, Department of Computer Science, University of British Columbia, 2000.

[18] L. Wall, T. Christiansen, and R. Schwartz. *Programming Perl*. O’Reilly & Associates, 2nd edition, 1996.

[19] D. Yellin and R. Strom. Interfaces, protocols, and the semi-automatic construction of software adaptors. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 176–190, 1994. ACM SIGPLAN.