Joinpoints as Ordered Events: Towards Applying Implicit Context to Aspect-Orientation

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

Abstract
Implicit context is a recently introduced mechanism for improving source code structure, making components more reusable, and making their systems more evolvable. It worries about eliminating locally-unneeded knowledge of external components (*extraneous embedded knowledge*, or *EEK*). Aspect-oriented programming (AOP) is a separation of concerns mechanism. It worries about separating and encapsulating functionality that crosscuts the components of a system. AspectJ is a prototype AOP language.

Both approaches can be seen to manipulate software at *joinpoints*, points in the execution or lexical description of that software. Joinpoints and the related code to execute at them can be considered as an event-based system. The exposure of events in AspectJ differs from that in implicit context—AspectJ considers these events as immediate and largely independent, while implicit context considers them as historic and interrelated through its *call history* concept. We give an example of a system in which limiting the description of the events of interest to those currently available in AspectJ results in an error-prone and hard-to-evolve implementation, but in which the use of call history would lead to a more reusable and robust implementation. We also briefly examine the issue of EEK and its removal within this example.

Keywords: Joinpoint, implicit context, aspect-oriented programming, AOP, AspectJ, extraneous embedded knowledge, EEK, event, call history, aspect.

1 Introduction

Many recent separation of concerns mechanisms involve composing pieces of source code, written in one or more languages, that describe different perspectives on (what would previously have been considered) the same components of a system. These pieces must be composed to form the fully functional components desired.

To describe how to compose the different pieces of source code, these mechanisms allow the programmer to statically refer to particular lexical points in the source code and/or to particular dynamic events; such points in the static source or dynamic execution are collectively referred to as *joinpoints*.[1] Common nameable joinpoints include the sending of calls to methods, the reception of such calls, the actual execution of a method, and the reading and writing of fields. Some of these mechanisms go further by exposing the state, or *execution context* that is available at a given joinpoint.

In aspect-oriented programming (AOP) [3], certain properties of the system, such as synchronization or distribution, are seen to *crosscut* the basic functionality. AOP has been partially realized in a prototype language built on top of Java, called AspectJ [2]. AspectJ allows a programmer to explicitly describe crosscutting functionality, or *aspects*, in localized code snippets.

Implicit context [5] is a recently-introduced mechanism and philosophy for structuring source code, which attempts to eliminate *extraneous embedded knowledge* (EEK) of the external world from components. Source code tends to be riddled with details, while an individual component should address a set of core concerns. To get such a component to interact with others to form a cooperative whole, that component ends up expressing knowledge of components that are external to it. For example, a component needs to know the precise interfaces to those external components that it must call, such as the names of particular classes or orders of parameters; these details are not inherent in the task that the component is designed to accomplish, and thus is EEK. Should the system require alteration such that this knowledge become invalid, then the component would require invasive modifications; such a process is difficult and error prone at a minimum.

Implicit context attempts to make components more reusable by allowing them to ignore the details of the environment in which they execute—they can each have their own *world view*. Conceptually, when messages pass into and out of a given component, those messages can be intercepted, changed, and rerouted as needed to translate between these

Position paper submitted to the Workshop on Advanced Separation of Concerns at the 23rd International Conference on Software Engineering, 2001.

[1] This term was coined by Bill Harrison at the AOP and Friends workshop at Xerox PARC in October 1996. The first written mention of the term appears to be in a technical report from Xerox PARC [4].

different world views; this process is called *contextual dispatch* and is performed at the *boundaries* between the differing world views. To understand the way in which a given message needs to be altered, contextual dispatch can utilize the *call history* of the system, to determine what messages have passed between components previously, and to locate the parameter and return values attached to these messages.

Concerns (in the sense of "features") and EEK are only partially-overlapping concepts: EEK often involves very fine-grained details too small to be considered concerns, while a component could contain multiple concerns that are important to it that would therefore not be EEK. Thus, aspect-orientation and implicit context are only partially-overlapping philosophies. Implicit context views the call history of a system to be an important, inherent feature. AOP says nothing about call history, but if crosscutting concerns were more separable through its use, it would be a useful AOP mechanism as well. In this paper, we examine how an AOP approach can be improved through the use of call history and, to a lesser extent, the removal of EEK. The question of how the full combination of implicit context and AOP would improve software is left to future work.

In Section 2, we describe how joinpoints can be viewed as events in the execution of a system, and how the provision of relative ordering information amongst these events, as exposed by implicit context, provides for more powerful use of the context of execution. We give a concrete example in Section 3 of a locking protocol in which such contextual information makes components more reusable and their system easier to evolve. Finally, we describe our conclusions in Section 4.

2 Joinpoints as Ordered Events

In AspectJ, aspects are applied to core source code by specifying the joinpoints at which the extra code should operate. In a language utilizing implicit context, one also specifies joinpoints at which extra code should operate.

An AspectJ joinpoint with aspect code to be attached to it can easily be seen as an event-based system: when the event described by the joinpoint occurs, the specified code (essentially, a callback) is executed. Pointcuts are a means of describing higher-level events in terms of primitive ones; we have described elsewhere how viewing high-level events as aggregations of primitive events is possible and useful [7]. Others have described how AspectJ's description of pointcuts is unsatisfactory as a simple event mechanism [1].

Consider the history of calls within a system: it can be represented as a directed tree. The nodes of this tree are joinpoints. Most of AspectJ's joinpoint designators permit access to nodes of this tree in isolation. Some of these designators, e.g., `cflow`, permit access to a particular ancestor within the current call stack. However, these are forward-looking: "while a call to method `foo` remains on the stack, look for joinpoints `bar` and `baz`."

Instead, implicit context (potentially) allows access to the full tree, providing an explicit ordering on the primitive events therein. Interesting states of the system can be described in terms of previous events and the ordering of them. We have described elsewhere how reasoning about events that have a relative order can be useful for software engineering tasks, such as solving performance problems or initially becoming familiar with the behaviour of a system [6]. Such ordered events provide a greater means to discriminate between joinpoints and to expose system state than is otherwise available.

3 A Resource Locking Protocol

The following code example is an abstraction of a system that could be used for performing remote method invocation or caching disk access. `Reader` objects contend for shared caches. If a cache is invalid, the `Reader` is responsible for creating a `TransmissionManager` that retrieves data and stores it to the cache. Once a cache is valid, the `Reader` simply retrieves the data in the cache and continues on to process it.

```
class Reader {
  private Cache cache;

  Reader(Cache cache) {
    this.cache = cache;
  }

  String read() {
    if(!cache.isValid())
      new TransmissionManager().
        downloadData(cache);

    String data = cache.getData();

    // Process the data
    // ...

    return data;
  }
}

class TransmissionManager {
  void downloadData(Cache cache) {
    // Get the data and store in cache
    // ...
  }
}

abstract class Cache {
  abstract boolean isValid();
  abstract String getData();
  // ...
}
```

Lock Requirements

Race conditions exist in this system. We want communication with the `TransmissionManager` to be minimized,

since that is the purpose in caching the data. Two or more `Readers` may have near-simultaneously failed the `cache.isValid()` test, and proceeded to request that the same data be downloaded. Each would then overwrite the cache.

On the other hand, the `cache.isValid()` test may have passed, but before the `Reader` is able to retrieve the data from the cache, the cache could become invalidated (through some process not shown).

To prevent both race conditions, we need each `Reader` to obtain a lock on the cache before the validity test is performed, and to keep it until the data is actually retrieved from the cache. Giving up the lock anywhere in between could allow the state of the cache to be changed in the meantime.

We also want to minimize the time that the cache is locked, since this resource is liable to become a bottleneck in the system otherwise. To permit maximum parallelism, multiple read accesses should be permitted to occur at once, and the cache should only be fully (write) locked if its data needs to be altered.

It would not be sufficient to define self-exclusive methods on the cache (through the `synchronized` keyword, for example) or mutually-exclusive sets of methods. A thread needs to ensure that a lock is maintained in a transaction-like fashion: not until the full task is completed should the lock be released or inconsistency could otherwise result.

We begin by providing a simple `Lock` aspect that allows a thread to obtain either a read or a read/write lock, to release the lock completely, or to downgrade a read/write lock to the weaker read lock:

```
abstract aspect Lock {
  synchronized void getReadLock() {...}
  synchronized void getReadWriteLock() {...}
  synchronized void releaseLock() {...}
  void onlyReleaseWriteLock() {...}
}
```

Now, we examine how we can weave this lock into our example through a minimalistic aspect-oriented approach, then demonstrate how a better AOP approach is achieved by removing some of the EEK present in the first approach, and finally, present how the extension of the second approach through the use of call history will make the system more robust.

A Minimalistic Aspect-Oriented Approach

Two aspects are needed. The first aspect is used to associate a unique lock instance with every cache.

```
aspect CacheLock extends Lock
  of eachobject(instanceof(Cache)) {
}
```

The second aspect enforces the protocol over each execution of `Reader.read()`. Three pieces of advice achieve this:

- The `before` obtains a read lock before `isValid()` is called on the cache. This lock needs to be held until after the data is actually read from the cache.
- The `after` releases the read lock after `getData()` is called on the cache.
- The `around` handles the situation where the cache is invalid, and data needs to be downloaded. To accomplish this, a write lock is needed. To avoid deadlock, the held read lock is released and a write lock is obtained. While waiting for the write lock, the cache may have been made valid by another thread, so the validity test is repeated. If the cache remains invalid, the data is actually downloaded. The write lock is then downgraded to a read lock.

```
aspect ReaderCacheLocking
  of eachcflowroot(ReaderCacheLocking.
                   entries()) {

  pointcut entries():
    receptions(String Reader.read());

  pointcut startLock(Cache cache):
    calls(boolean cache.isValid());

  pointcut endLock(Cache cache):
    calls(boolean cache.getData());

  pointcut download(Cache cache):
    calls(void TransmissionManager.
              downloadData(cache));

  boolean hasReadLock = false;
  boolean hasWriteLock = false;

  before(Cache cache): startLock(cache) {
    if(!hasReadLock) {
      CacheLock lock =
        CacheLock.aspectOf(cache);
      lock.getReadLock();
      hasReadLock = true;
    }
  }

  after(Cache cache): endLock(cache) {
    CacheLock lock =
      CacheLock.aspectOf(cache);
    hasReadLock = false;
    hasWriteLock = false;
    lock.releaseLock();
  }

  around(Cache cache) returns void:
    download(cache) {
    if(!hasWriteLock) {
```

```
      CacheLock lock =
        CacheLock.aspectOf(cache);

      lock.releaseLock();
      hasReadLock = false;
      hasWriteLock = false;

      lock.getWriteLock();
      hasWriteLock = true;
      hasReadLock = true;

      if(!cache.isValid())
        proceed(cache);

      lock.onlyReleaseWriteLock();
      hasWriteLock = false;
    }
    else {
      // Uh-oh, a recursive write is bad
      throw new RuntimeException();
    }
  }
}
```

Removing Some EEK

While the aspects just described get their immediate job done, problems remain. A `CacheLock` instance does not know who possesses it; it is therefore up to each `ReaderCacheLocking` instance to ensure that it actually obtains and releases the appropriate lock. Should an engineer alter the system such that the cache is accessed in other places, they would need to realize that the locking protocol needed to be enforced within other aspects; failing to create such an aspect would violate the synchronization constraints on the cache. Furthermore, if the engineer gets and releases locks in the wrong sequence within the aspect, the synchronization can also be easily violated. Therefore, we want to alter these aspects to separate out direct enforcement of the synchronization constraints, manifested in the from the `hasReadLock` and `hasWriteLock` attributes, particular protocol needs in the case of `Reader`.

To enable this separation, we take advantage of the fact that we know that locks are held on a per-thread basis, and that we can always determine the current thread. We have added four pieces of advice to `CacheLock`: one ensures that a read lock is held by the current thread before any read operations execute, one ensures a write lock is held before any write operations execute, and the other two ensure that a lock is actually held if an attempt is made to release one.[2] The `ReaderCacheLocking` aspect is simplified (not shown) by removing all mention of the `hasReadLock` and `hasWriteLock` attributes, along with any statements that utilize them.

[2] Any of these pieces of advice could throw exceptions instead (or in addition) if so desired.

```
aspect CacheLock extends Lock
  of eachobject(instanceof(Cache)) {
  pointcut read():
    receptions(boolean isValid()) ||
      receptions(String getData()) ||
        /* other read methods */;

  pointcut write():
    receptions(/* any write methods */);

  pointcut getWriteLock():
    receptions(void getReadWriteLock());

  pointcut releaseWriteLock():
    receptions(void releaseLock()) ||
    receptions(void onlyReleaseWriteLock());

  pointcut releaseReadLock():
    receptions(void releaseLock());

  pointcut getReadLock():
    receptions(void getReadLock()) ||
      getWriteLock();

  HashSet readLocks = new HashSet();
  Thread writeLock = null;

  around() returns Object: read() {
    boolean mustRelease = false;
    Thread thread = Thread.currentThread();

    if(!readLocks.contains(thread)) {
      mustRelease = true;
      getReadLock();
      readLocks.add(thread);
    }

    Object obj = proceed(cache);

    if(mustRelease) {
      readLocks.remove(thread);
      releaseLock();
    }

    return obj;
  }

  around() returns Object: write() {
    boolean mustRelease = false;
    boolean hadReadLock = false;

    Thread thread = Thread.currentThread();

    if(writeLock != thread) {
      mustRelease = true;

      if(readLocks.contains(thread)) {
        hadReadLock = true;
        readLocks.remove(thread);
        releaseLock();
```

```
      }

      getReadWriteLock();
      readLocks.add(thread);
      writeLock = thread;
    }

    Object obj = proceed(cache);

    if(mustRelease) {
      writeLock = null;

      if(hadReadLock)
        onlyReleaseWriteLock();
      else {
        readLocks.remove(thread);
        releaseLock();
      }
    }

    return obj;
  }

  around() returns void: releaseReadLock() {
    Thread thread = Thread.currentThread();

    if(readLocks.contains(thread)) {
      readLocks.remove(thread);
      proceed();
    }
  }

  around() returns void:
    releaseWriteLock() {
    Thread thread = Thread.currentThread();

    if(writeLock == thread) {
      writeLock = null;
      proceed();
    }
  }
}
```

The `hasReadLock` and `hasWriteLock` attributes are EEK within the `ReaderCacheLocking` aspect. It is not clear that one would *a priori* consider them to be part of a concern that is separate from the acquisition and release of a lock on the cache, which are central to this aspect. But the philosophy of the identification and removal of EEK do highlight these candidates for separation, even without an analysis of the effects of their presence. Other EEK is present, such as explicitly accessing other aspect instances, but is not removable through AspectJ as it currently exists.

This solution is getting a bit ugly, though. The `ReaderCacheLocking` aspect needs to explicitly worry about each thread that has to access it and to keep track of each: more EEK! Since the attributes recording the locks can be accessed by multiple threads at once, the question of synchronization within the aspect arises—and it is not so obvious where such synchronization should go. Finally, which advice is responsible for adding and removing threads from the lock collections? At the moment, all do, but this could easily be wrong. Does advice apply to other advice in the aspect, and if so, in what order? The answers, though likely to be defined, are murky and therefore cause the behaviour of this code to be difficult to understand. Utilizing call history can help clear things up.

Adding-in Call History

Modifying the `CacheLock` aspect to utilize call history would allow us to simplify and clarify the purpose of this aspect. In the modified aspect below, we have posited call history constructs that could fit into the existing framework of AspectJ; the constructs are based on those being investigated in our work on implicit context.

These constructs permit one to test the order of events in the call history. Now, instead of checking to see if some attribute contains the current thread, we simply define that the current thread has a read lock if its last read lock release occurred prior to its last read lock get.[3] The pointcut definitions remain identical to those in the previous example.

```
aspect CacheLock extends Lock
  of eachobject(instanceof(Cache)) {
  pointcut read():
    receptions(boolean isValid()) ||
      receptions(String getData()) ||
        /* other read methods */ ;

  pointcut write():
    receptions(/* any write methods */);

  pointcut getWriteLock():
    receptions(void getReadWriteLock());

  pointcut releaseWriteLock():
    receptions(void releaseLock()) ||
    receptions(void onlyReleaseWriteLock());

  pointcut releaseReadLock():
    receptions(void releaseLock());

  pointcut getReadLock():
    receptions(void getReadLock()) ||
      getWriteLock();

  condition hasReadLock():
    moreRecent(getReadLock(),
               releaseReadLock());

  condition hasWriteLock():
    moreRecent(getWriteLock(),
               releaseWriteLock());

  around(Cache cache) returns Object:
```

[3] If neither call has happened, `moreRecent` would return `false`.

```
    read() &&
    !hasReadLock() {
    getReadLock();
    Object obj = proceed();
    releaseLock();
    return obj;
  }

  around() returns Object:
    write() &&
    !hasReadLock() {
    getReadWriteLock();
    Object obj = proceed();
    releaseLock();
    return obj;
  }

  around() returns Object:
    write() &&
    hasReadLock() &&
    !hasWriteLock() {
    releaseLock();
    getReadWriteLock();
    Object obj = proceed();
    onlyReleaseWriteLock();
    return obj;
  }

  around() returns void:
    releaseReadLock() &&
    !hasReadLock() {
  }

  around() returns void:
    releaseWriteLock() &&
    !hasWriteLock() {
  }
}
```

We have split the advice to be applied to the `write()` pointcut into two pieces of advice: one deals with the condition when no read lock is possessed (and hence, no write lock either), and the other deals with the condition when only a read lock is possessed. Such a separation could be foregone if one desired to and if the condition construct were usable within the body of an advice.

Since there is no state to be maintained within the `CacheLock` aspect, we do not have to be concerned about synchronization. The purpose and behaviour of each of the five pieces of advice are much clearer now, as each applies to a well-defined condition of the system.

4 Conclusions

We have described how implicit context's notions of EEK and call history can make an aspect-oriented program more robust. Call history can be seen as an extension to the expressiveness of joinpoint specifications.

Applying the philosophy of removing EEK allowed us to enforce the locking protocol. If the system utilizing caches were extended, the protocol would still be followed even if the extension did not perform properly. Some EEK still remained in this solution, notably, as the `ReaderCacheLocking` aspect had to explicitly retrieve the pertinent `CacheLock` aspects, but removing such EEK would require more radical additions to AspectJ that cannot be described in the space available. Applying call history to the resulting `CacheLock` aspect made it easier to understand, and subtle worries over ordering of advice and synchronization vanished.

Call history is a mechanism from which AOP would benefit directly. The other parts of implicit context, such as world views and greater removal of EEK, do not likely mesh with the philosophy of AOP, but the combination of the two approaches should be investigated in the future.

Acknowledgements

This locking protocol example is based on a discussion between Rich Price and Jim Hugunin on the `users@aspectj.org` mailing list. AspectJ is a trademark of Xerox Corporation.

REFERENCES

[1] B. de Alwis, S. Gudmundson, G. Smolyn, and G. Kiczales. Coding issues in AspectJ. In *Workshop on Advanced Separation of Concerns in Object-Oriented Systems at the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Minneapolis, USA, 16 Oct. 2000. Position paper.

[2] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming*, 2001. To appear.

[3] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97—Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pp. 220–242, Jyväskylä, Finland, 9–13 June 1997.

[4] A. Mendhekar, G. Kiczales, and J. Lamping. RG: A case-study for aspect-oriented programming. Technical Report SPL97-009 P9710044, Xerox Palo Alto Research Center, Palo Alto, USA, Feb. 1997.

[5] R. J. Walker and G. C. Murphy. Implicit context: Easing software evolution and reuse. In *Proceedings of the ACM SIGSOFT Eighth International Symposium on the Foundations of Software Engineering*, pp. 69–78, San Diego, USA, 8–10 Nov. 2000.

[6] R. J. Walker, G. C. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak. Visualizing dynamic software system information through high-level models. In *OOPSLA '98 Conference Proceedings: Object-Oriented Programming, Systems, Languages, and Applications*, pp. 271–283, Vancouver, Canada, 18–22 Oct. 1998.

[7] R. J. Walker, G. C. Murphy, J. Steinbok, and M. P. Robillard. Efficient mapping of software system traces to architectural views. In *Proceedings of CASCON 2000*, pp. 31–40, Mississauga, Canada, 13–16 Nov. 2000.