

# Computer Sciences Department

Code Sandwiches

Matt Elder

Steve Jackson

Ben Liblit

Technical Report #1647

October 2008



# Code Sandwiches

Matt Elder, Steve Jackson, and Ben Liblit  
University of Wisconsin–Madison

## Abstract

A code sandwich consists of *before*, *after*, and *body* code. Typically, the *before* code makes some change, the *body* code works in the context of the change, and the *after* code undoes the change. A code sandwich must guarantee that its *after* code will execute if its *before* code has executed, even if exceptions arise. This pattern is common to many programming situations, and most modern languages have some language-level support for expressing it.

We survey support for code sandwiches in several programming languages and proposed language extensions. We explain why such support can improve a program, consider related features, and discuss desirable properties that a language can provide its programmers. We relate these properties to Jyro, our code sandwich extension to Java. We examine two large, mature open-source programs, find numerous sandwich-related bugs and readability issues, and demonstrate how they might be avoided using our Jyro implementation.

## 1 Introduction

Most programmers have encountered a situation like the following:

```
1 // L is a mutex lock
2 lock( L );
3 /* do work */
4 unlock( L );
```

In the above code snippet, lines 2 and 4 are semantically linked: they perform dual operations on a single resource. In any well-behaved program, these operations occur only in ordered pairs. A lock must always be followed by an unlock, and an unlock may only occur after a corresponding lock. However, the link between these operations is invisible to the compiler. The programmer must ensure that such implicitly-linked operations occur at correct times, in a correct order.

This responsibility is complicated by the code represented by line 3 in the above example. Even a well-behaved program may do an arbitrary amount of work between calls to `lock` and `unlock`. The two calls may be separated visually by many pages of code. Worse, the intervening code may contain control flow (such as a `break` statement) that bypasses the call to `unlock`, creating an opportunity for the program

to miss the unlocking step. Furthermore, if the language supports exceptions, this problematic control flow may be invisible to the programmer.

Such situations arise whenever a program manipulates shared resources. APIs for locks, sockets, files, or database connections may require a program to explicitly close or release a resource that it previously acquired. In a language without garbage collection, the programmer is responsible for allocating memory before its use and releasing it after its use. In general, a variety of programming tasks call for a program to make a change, operate in the context of that change, and then undo the change. We call such situations *code sandwiches*. Section 2 defines this term formally and gives additional background, while Section 3 considers related prior work.

Several modern languages have evolved programming mechanisms to help programmers write safe, defensive code in the situations described above. These typically allow a programmer to make explicit the connection between code that makes a change and code that reverses the change. In Section 4, we review such mechanisms in several languages and proposed language extensions. Each has its strengths and weaknesses; we examine the trade-offs in Section 5. Our goal is to characterize the design space for such language features, in the hope that future language designers will make design choices with open eyes. We also hope to provide a common basis for discussing these mechanisms.

As an example of “sandwich-aware” language design, we have extended Java with explicit support for code sandwiches. We describe our language extension, Jyro, in Section 6. Section 7 relates our experience using a prototype Jyro implementation to fix numerous bugs and readability issues in complex, real-world applications. Section 8 concludes.

## 2 Code Sandwiches

Informally, think of a code sandwich as three “pieces” of code: code that makes a change, code that operates in the context of that change, and code that reverses the change. More formally,

**Definition 1** A **code sandwich** is a triple formed of three segments of source code, called before, body, and after, with two properties:

- body code executes between the before and after code.
- Correct program behavior requires that, for any before code that executes, the associated after code also executes.

In the terminology of control flow analysis, we say that *after* code must *postdominate* its associated *before* code. If a sandwich does not meet this condition, we call it defective:

**Definition 2** If there exists a path through the body of a code sandwich that causes the after code to be skipped, then the program has a **code sandwich defect**.

Code sandwiches appear in many programming situations. Several common examples relate to the acquisition and release of scarce resources, such as locks, file descriptors, or socket connections. In more general cases, any temporary change of program state may require a code sandwich. For example, a GUI-based program may temporarily ignore user inputs, or an OS kernel may temporarily disable hardware interrupts. Failure to restore earlier state in these cases will cause serious bugs.

Defective code sandwiches arise most frequently in the presence of exceptions and their associated invisible control flow. Indeed, special language features to manage code sandwiches arise chiefly in languages that support exceptions (see Section 4).

However, exceptions are not the only cause of defective code sandwiches. Whenever changes are made to *body* code, new control paths may arise that bypass the *after* code. In the simplest case, a maintainer need only add a `return` statement to a sandwich's *body* to introduce a new defect, which may lead to silent errors. When the *body* code is large and *before* and *after* are widely separated, such mistakes can be hard to detect visually.

Errors caused by defective sandwiches are not always catastrophic. A resource leak will rarely crash a program, though it may cause a slowdown [24]. Thus, such bugs may be difficult to find with standard debugging tools. In some cases, specialized tools may be used to detect these defects statically [19]. However, we prefer to give the programmer some reliable way to avoid mistakes in the first place.

### 3 Related Work

The SABER analysis of Reimer et al. [19] detects several types of faults, including code sandwich faults. Sandwich defects are violations of “Must call X after Y” rules. SABER finds such defects in three of the four stable commercial Java applications they test, heightening our concern that poor language support breeds sandwich-related bugs.

Weimer and Necula [25] create an analysis to find resource management bugs in the presence of exceptions. The analysis looks for cases where exceptional control flow can cause a resource, such as a file handle or socket connection, to be leaked. They run their analysis across more than two dozen mature Java applications and find exception-handling errors in all but one. In response to these results, the authors propose a Java language extension, compensation stacks, which we discuss further in Section 4.9.1. This extension is partly modeled on a theory of compensating transactions in databases [14].

AspectJ and other aspect-oriented programming (AOP) languages let the programmer define executable advice on sets of method calls, constructors, field accesses, and similar actions [12, 13]. One can implement a superset of the code sandwich support we discuss using the `before`, `after`, and `around` advice forms. However, when a code sandwich is local to a particular scope, object, or context, such use is an abuse of a system meant to centralize specification of cross-cutting concerns. For example, sandwiches often describe opening and closing files, but when writing to a file is the primary purpose of a function, treating the matched open/close operations as an aspect may be inappropriate. Rich language support for code sandwiches simplifies writing aspects just as it simplifies writing traditional methods and objects. Thus we suggest

that AOP and language support for code sandwiches are mutually compatible but mostly orthogonal.

## 4 Overview of Existing Techniques

A language may provide three special features to help programmers avoid code sandwich defects:

- **Inevitability:** The language may provide a means to make *after* code inevitable once execution reaches a certain program point. This can make sandwich code “correct by construction”: a programmer using this feature can be certain that *after* code will never fail to run.
- **Syntactic linkage:** The language may provide a mechanism to explicitly associate *before* and *after* code, usually with special syntax. Such a mechanism promotes program understanding and eases maintenance. The link between *before* and *after* code becomes visible to both the programmer and the compiler.
- **Encapsulation:** The language may provide a means to encapsulate a common code sandwich pattern and reuse it.

In this section, we survey how a variety of modern languages offer one or more of these features.

### 4.1 C++: Destructors and RAII

C++ associates constructor and destructor functions with class definitions. When a stack-allocated C++ object falls out of lexical scope, its destructor is called immediately. This is most obviously useful as a means to release memory explicitly allocated in the class’s constructor. However, destructors can also guarantee release of other resources. For example, the mutex locking code from Section 1 might be changed to the following:

```
1 // L is a mutex lock
2 {
3     scoped_lock S( L );
4     /* do work */
5 }
```

Here, `scoped_lock` is a class whose constructor locks the given mutex and whose destructor unlocks the same. Thus `L` is locked on line 3 and unlocked on line 5 when the object `S` goes out of scope [23]. This idiom for resource management is referred to as *Resource Acquisition Is Initialization*, or RAII.

RAII provides the three language features discussed above. In typical usage, a sandwich’s *before* and *after* code reside in the constructor and destructor of a single class. This creates a clear syntactic link between the two. The language guarantees that the destructor will run when control exits its containing scope, even if this happens as a result of a propagating exception. Thus, the *after* code is inevitable. Lastly, because

RAII builds upon classes, C++ provides strong support for encapsulating sandwiches as reusable components.

(Variations on this form of syntactic linkage are possible, though rare. A programmer could put *before* code elsewhere, and only use the *after*-inevitability guarantees provided by the destructor.)

Boehm [2] points out that the RAII idiom is restricted to languages where destructors run at well-defined times. It does not translate cleanly to garbage-collected languages like Java, where finalizers run at the whim of a separate garbage collection thread and therefore may be significantly delayed. Some kinds of cleanup code may endure garbage collection delays without serious problems. For example, a cleanup routine that closes a temporary file may not need to run as soon as possible after the program finishes interacting with the file. For thread synchronization via mutex unlocking, however, cleanup code must run immediately.

#### 4.1.1 C++: Special Cases

Several standard C++ classes constitute resource management sandwiches, all of which can be built using the language primitives already described. The `auto_ptr` template class behaves like a pointer but also encapsulates unique ownership of allocated memory. This memory is freed in `auto_ptr`'s destructor unless ownership has already been transferred elsewhere. Similarly, file stream classes open the file named in the constructor, then close it in the destructor.

File streams demonstrate that a sandwich encapsulation may provide nontrivial functionality (e.g., formatted input and output routines) for use within the sandwich body. A sandwich encapsulation need not be limited to providing only *begin* and *end* code. Because this additional functionality is only available within the scope of the body, run-time errors such as writing to a closed file can be prevented at compile time. However, as we will see with Python (Section 4.5), alternate designs can provide a scope-limited, sandwich-managed object which is distinct from the sandwich proper.

## 4.2 Java: The `finally` Keyword

Java specifies `try/catch` blocks for exception-prone code [10]. A `try` block may be followed by an optional `finally` block. The language guarantees that the code in the `finally` block will execute regardless of how the `try` block exits (either normally or through an exception).

Our running mutex example might be expressed in Java as:

```
1 // L is a mutex lock
2 lock( L );
3 try {
4     /* do work */
5 } finally {
6     unlock( L );
7 }
```

```

boolean locked = false;
Lock commitLock;
try {
    locked = commitLock.obtain( /* args */ );
    return /* complex expression */ ;
} finally {
    if (locked) {    commitLock.release();    }
}

```

Figure 1: Java's try/finally in practice. Example taken from Lucene.

The *before* code (here, the lock operation on line 2) might appear either within the try block or above it, depending on the situation. In either case, there is no explicit syntactic linkage between *before* and *after*, and no convenient means for encapsulation or reuse of common patterns. The finally block only provides inevitability.

Figure 1 shows an example in which finally is used to release acquired resources. Note the use of the flag value `locked`, which indicates the success of the exception-prone call to `commitLock.obtain`. In this example, the *after* code first checks the flag to determine if the `release` method should be called.

#### 4.2.1 Java: Special Cases

Java's `synchronized` keyword handles the common case of low-level mutex locking for thread synchronization. This keyword provides a syntactic convenience for lexically scoped, exception-safe mutex locking. Manual locking, as in the example we show above, is still necessary for programmers interacting with the lock structures in the `java.util.concurrent` package, or with custom lock structures.

The Java privileged code API [8] provides a mechanism for code to temporarily increase its security privileges. Privileged actions are often specified as anonymous inner classes, using the following pattern:

```

AccessController.doPrivileged(
    new PrivilegedAction() {
        public Object run() {
            /* work to do while privileged */
        }
    });

```

This is an example of library-based support for a common code sandwich pattern. In this case, *before* is the action of enabling privileged mode, and *after* is the action of disabling it. The user of the `java.security` library specifies the *body* code in the `run` method of a `PrivilegedAction` object.

In older versions of the JDK, privileged actions were started and stopped using explicit calls to `beginPrivileged` and `endPrivileged`. The prescribed pattern for using these two functions safely was a try/finally block [9]. Changing to

`doPrivileged` simplifies JIT compilation by making privilege boundary crossings easier to identify. Unfortunately, inner classes impose verbose syntax and variable-access restrictions that can make using the new API cumbersome [16].

### 4.3 D: Scope-Guarded Statements

D is a language proposed as a replacement for C and C++ in the systems and applications domain [5]. It has C-like syntax, and has support for both RAII patterns and `try/finally` constructs. Furthermore, D has a third kind of code sandwich support: the `scope guard` syntax. A `scope-guarded` statement takes one of three forms:

```
scope (exit)    [statement]
scope (success) [statement]
scope (failure) [statement]
```

In each case, the given `statement` is not executed until control exits from the containing lexical scope. A statement guarded by `success` is only executed if the containing scope exits normally (i.e., not because of an exception). Conversely, the `failure` guard specifies that the statement should only be executed if the containing scope exits due to a propagating exception. The `exit` guard executes the statement in either case.

Using scope guards, our earlier example becomes

```
1 // L is a mutex lock
2 {
3     lock( L );
4     scope (exit) unlock( L );
5     /* do work */
6 }
```

The mutex will be unlocked when control reaches line 6.

Scope-guarded statements execute in reverse order from when they were encountered. This corresponds to the behavior of C++ destructors: when several C++ objects go out of scope at the same time, they are destructed in reverse order of their declarations.

Use of scope guard statements can obviate the need for flags that arises in the context of `finally` blocks (see Figure 2). In addition, the `failure` and `success` guards allow cleanup statements to be sensitive to the success or failure of *body* code.

The scope guard syntax provides an inevitability mechanism for *after* code. It does not create an explicit syntactic linkage between *before* and *after* code. However, the programmer is free to place the *before* and *after* code next to each other to make the association clear to the code reader. This contrasts with `try/finally` constructs, where *before* and *after* code are visually separated by *body* code. Scope guards do not provide encapsulation.

In some cases, the scope guard pattern can be implemented using the existing features of another language. Nasonov [17] proposes a library-based implementation of `scope (exit)` as an extension of the C++ Boost libraries. As of this writing, the review status of the extension is “pending.” [3]



```

Lock commitLock;
{
    commitLock.obtain( /* args */ );
    scope (exit) commitLock.release();
    return /* complex expression */ ;
}

```

Figure 2: Scope guard statements in D. Example rewritten from Figure 1.

#### 4.4 C#: The using Statement

C# finalizers<sup>1</sup> run under control of the garbage collector, not the programmer, and therefore are inappropriate for encapsulating sandwich *after* code. Instead, C# offers the `System.IDisposable` interface and the `using` statement to define and invoke cleanup actions:

```

1 // L is a mutex lock
2 using (ScopedLock lock =
3     new ScopedLock( L )) {
4     /* do work */
5 }

```

Here `ScopedLock` is a class whose constructor locks the given mutex. `ScopedLock` must implement the `System.IDisposable` interface, which requires a single void `Dispose()` method. The `using` statement binds a local variable to an `IDisposable`-implementing value and invokes its `Dispose` method when the `using` block completes. In this example, `ScopedLock.Dispose` should unlock the mutex. The `using` statement will invoke `lock.Dispose` on line 4, when the block terminates.

C# `using` statements are exception-safe: the `Dispose` method will be called regardless of how the `using` statement terminates. In fact, the C# language specification [6] clarifies the behavior of `using` by rewriting examples into a `try` blocks with `finally` clauses similar to that shown in Figure 1.

As a syntactic convenience, a `using` statement may bind several variables in sequence; this is equivalent to nested `using` statements. An anonymous `IDisposable` value may also be used if no bound name is required:

```

// L is a mutex lock
using (new ScopedLock( L )) {
    /* do work */
}

```

It is worth noting that `Dispose` methods and finalizers have no special relationship imposed at the language level. Neither implicitly calls the other, and there is no guarantee

<sup>1</sup>Earlier versions of the C# specification used the term “destructor.” This has since been abandoned due to confusion with C++ destructors, which have similar syntax, but run at deterministic times [6].

that a `using`-bound value will be garbage-collected promptly when the `using` block completes. Coordinating resource release between finalizers and `Dispose` methods is entirely the programmer's responsibility.

In the language of code sandwiches, we can say that C# provides inevitability and encapsulation, and permits but does not enforce syntactic linkage. The programmer may choose to put the *before* code inside the constructor of the `IDisposable` object, as in our `ScopedLock` example. However, since a single `IDisposable` object may be used with multiple `using` blocks, the programmer may prefer to put *before* code somewhere other than the constructor.

## 4.5 Python: Context Managers

Python 2.5 introduces a type of object called a *context manager* and an associated keyword `with`. This new feature “allows common `try...except...finally` usage patterns to be encapsulated for convenient reuse.” [20] A context manager is an object that defines a *before* function named `__enter__` and an *after* function named `__exit__` [21]. Context managers are invoked by the `with` statement as follows.

```
# L is a mutex lock
with ScopedLock( L ):
    body
```

Here `ScopedLock` is assumed to be the constructor for a context manager for locks. Code is executed in the expected order: first `__enter__` executes, then the *body* code, then `__exit__`. The exit function is guaranteed to run even if the *body* code throws an exception.

Python context managers are similar to C# `IDisposable` objects. Both provide inevitability: they guarantee that their *after* code always executes. Both provide encapsulation by means of specially handled objects. Python provides a more obvious means of syntactic association via the pairing of `__enter__` and `__exit__`.

### 4.5.1 Context Managers as Single-Value Generators

Python defines an additional shortcut for creating context manager objects using generator functions. A generator function that `yields` exactly once can be treated as a context manager. For example, the following is a complete implementation of `ScopedLock` in a single function:

```
1 @contextmanager
2 def ScopedLock(mutex):
3     mutex.lock()
4     yield
5     mutex.unlock()
```

The `@contextmanager` decorator on line 1 converts `ScopedLock` from being a generator function that yields a single value into a complete object with `__enter__` and `__exit__` methods. The `__enter__` method includes all execution before the

`yield`, while `__exit__` performs all code after the `yield`. From the perspective of a library developer creating such a sandwich, `yield` marks the location where a caller's *body* code would be injected.

This provides a novel way to share state between *before* and *after* code: the two code segments share the local context of a single function, with no need for private object fields or global variables.

#### 4.5.2 Scoped Name Binding

An extended `with` syntax offers scoped name binding similar to, but more general than, that offered by C#:

```
# L is a mutex lock
with ScopedLock( L ) as crit:
    body
```

This binds `crit` to the value returned by `ScopedLock.__enter__` (or provided by `yield`) while the *body* runs. This bound value need not be the context manager itself: it can be any arbitrary value. For example, a file context manager might create and return an opened file object from `__enter__` for use within the body, with the file and the file context manager being distinct objects. C# can only bind a name to the `IDisposable` object itself, and therefore would require that the file context manager and file be one and the same.

## 4.6 Common Lisp

The code sandwich problem takes on a different aspect in a functional language. Programmers may find it convenient to encapsulate *before*, *after*, or *body* code in anonymous functions or closures, depending on the features provided by their language.

Common Lisp is a functional language with anonymous functions, closures, and an integrated macro system. It gives the programmer power sufficient to write custom flow control.

Standard in Common Lisp is the `unwind-protect` operator. This operator takes one or more Lisp forms as arguments. The first form is the *protected form*, and every subsequent form is a *cleanup form*. When the protected form exits, no matter the cause of exit, the cleanup forms will run. The semantics of `unwind-protect` do not guarantee that every cleanup form will run if one terminates abruptly; this guarantee requires nested `unwind-protects` [18].

```
(unwind-protect
 (progn
  (obtain-lock lock)
  (do-work))
 (release-lock lock))
```

Since Common Lisp permits macro definitions, such cases are often handled by specific macros exported by the relevant library. For example, `with-open-file`

opens a file, invokes user actions, and then closes the file after those actions, in the manner of `unwind-protect`. So, acquiring a lock through some library may be as simple as

```
(with-lock (lock)
  (do-work))
```

## 4.7 O’Caml

O’Caml admits both functional and object-oriented coding paradigms. It does not have explicit language support for code sandwiches. However, we can develop a library-based pattern for code inevitability as follows:

```
1 let doWithCleanup after body =
2   let result =
3     try
4       body ()
5     with problem ->
6       after (Some problem);
7       raise problem
8   in
9   after None;
10  result
```

The above defines a function `doWithCleanup` that takes two arguments: an *after* function and a *body* function. Any exceptions raised while calling *body* are temporarily caught (on line 5) and then re-raised after the *after* call finishes. If no exceptions arise, then the *after* call runs on line 9 after the *body* completes. Calls to *after* include the exception being thrown, if any, allowing specialized cleanup depending on the outcome.

We can specialize this generic function to create a code sandwich wrapper for the mutex locking pattern:

```
let withLocked mutex =
  Mutex.lock mutex;
  let after _ = Mutex.unlock mutex in
  doWithCleanup after
```

Note the use of function currying in this construct. By placing the *after* argument first when defining `doWithCleanup`, we allow specialized wrappers such as `withLocked` to provide cleanup code while still leaving the sandwich *body* unspecified. Under this arrangement, we can pass any *body* function to the return value of `withLocked L`, where `L` is a mutex, and this *body* function will be called with the lock held.

## 4.8 Scheme

Scheme is a dialect of Lisp with support for continuations. A procedure call's *dynamic extent* is the time between when it is entered and when it completes. Continuations allow calls to be reentered after they return, so a call's dynamic extent may include more than one span of time.

Such behavior complicates Scheme's support for code sandwiches. Since R5RS [11], Scheme has included the `dynamic-wind` operator. This operator can directly wrap locking and unlocking functions:

```
(dynamic-wind
  (acquire-lock lock)
  (do-work)
  (release-lock lock))
```

The `dynamic-wind` operator ensures that a call to `before` precedes every entry to the dynamic extent of `body`, and a call to `after` follows every exit [22]. In the lock example, `dynamic-wind` provides mutual exclusion even if `do-work`'s dynamic extent is reentered. Since mutual exclusion is often needed, the author of a locking library is likely to export an equivalent macro, just as in Common Lisp.

## 4.9 Proposed Java Extensions

### 4.9.1 Compensation Stacks

Weimer and Necula perform path analysis on a large corpus of Java code and show that resource handling bugs are common. They then propose a novel language extension as a solution [24, 25]. Their system allows the programmer to describe a *compensation* for a segment of code. A compensation is a set of statements that reverses the effect of the code it is attached to, such as by releasing a lock or closing a file.

Compensations are stored in special structures called *compensation stacks*, which are first-class objects in the extended Java language. Compensations stored in a stack are executed when a `run` function is called on the stack. Compile-time analysis ensures that no compensation stack goes out of scope unless all its compensations have run, thereby ensuring that no action is taken without its associated compensation eventually executing.

Compensation stack use is as follows:

```
// L is a mutex lock
CompensationStack S =
  new CompensationStack();
try {
  compensate { lock( L ); }
  with      { unlock( L ); } using (S);
  /* do work */
} finally { S.run(); }
```

The compensation stack approach differs from the approaches presented above in that a compensation stack is not inherently tied to the call stack or to scope nesting structure. Being objects in the language, compensation stacks may be passed to or returned from functions, and cleanup actions may be stored into them in any context. The potential power of this approach is discussed in Weimer’s original papers; we note it here as a potential point in the design space.

Weimer’s proposal includes syntactic sugar to simplify the case where lexically scoped cleanups are all that is desired. Thus, our above example is more verbose than strictly necessary.

#### 4.9.2 Java with Closures

Bracha et al. [4] propose to extend the Java language with explicit closures. This would provide another way to express code sandwiches in Java. Following a pattern similar to our O’Caml pattern above, we arrive at the following function for exception-transparent locking, which is very similar to one suggested by Bracha et al. [4]:

```
public static
<T, throws E extends Exception>
T withLock(Lock L, {=>T throws E} body)
  throws E {
    L.lock();
    try {
        return body.invoke();
    } finally {
        L.unlock();
    }
}
```

Here, *before* and *after* share the body of a single function, with inevitability achieved through the use of a `try/finally` block. The function may then be called, with the second argument being a closure representing the *body* of the code sandwich:

```
withLock(L, {=>
    /* body code */
});
```

In this example, the code sandwich pattern is encapsulated within a function. The *before* and *after* code do not have explicit syntactic linkage, but they are near enough to one another for the programmer to see their relationship easily.

#### 4.10 C: GDB Cleanup Chains

Although C has no exceptions, Section 2 notes that any non-sequential control flow can create code sandwich defects. Furthermore, C does offer a low-level mechanism for interprocedural control flow: `set jmp` and `long jmp`. A call to `set jmp` preserves the current stack context and registers in a program-supplied buffer. Passing the same

buffer in a later call to `longjmp` restores this stack context, jumping back to the earlier `setjmp` call even across function boundaries. One common use of `setjmp/longjmp` is to return execution to a top-level event dispatch or command processing loop. The application calls `setjmp` at the top of this loop. If processing of an event or command fails, the application uses `longjmp` to abandon the current task and quickly return to the dispatch loop for the next operation.

When used carefully and correctly, `setjmp` approximates `try/catch` while `longjmp` approximates `throw`. However, there is nothing to approximate `finally`, so resource leaks and other code sandwich defects seem unavoidable in code that uses this style. Avoiding leaks requires a combination of library support and careful adherence to coding guidelines.

The GNU Debugger (GDB) illustrates the extraordinary effort needed to get C sandwiches right. GDB uses `setjmp/longjmp` to return to its command processing loop as described above. Cleanup actions that should be performed when returning to this loop are collected in *cleanup chains*, which behave as stacks. Typical cleanup actions include releasing allocated memory and closing opened files. If GDB used mutex locks, we might find the following code:

```
1 // L is a mutex lock
2 lock( L );
3 cleanup *old = make_cleanup( unlock, L );
4 /* do work */
5 do_cleanups( old );
```

The `make_cleanup` call on line 3 records the old depth of a hidden global chain (stack) of cleanup actions. It also pushes an additional cleanup action onto this chain by providing a function handle (`unlock`) and an argument to be passed to that function (`L`). If the sandwich *body* code on line 4 completes without error, then the `do_cleanups` call on line 5 executes all saved cleanup actions down to the old cleanup chain depth recorded earlier. A distinct `do_cleanups` call in the main command processing loop (not shown) takes care of running cleanup actions after a `longjmp`.

Cleanup chains are GDB-specific. Other applications may differ, but would need a similar mechanism if they intended to use `setjmp/longjmp` in a similar manner. Cleanup chains provide inevitability assuming that the top-level dispatch loop calls `do_cleanups` correctly. Syntactic linkage is implicit, much like D's scope guards: programmers may put *before* and *after* code next to each other, such as on lines 2 and 3 of the above example, but there is no syntactic requirement to do so. Cleanup chains offer no encapsulation.

## 4.11 L<sup>A</sup>T<sub>E</sub>X

Though not a general-purpose programming language, the L<sup>A</sup>T<sub>E</sub>X markup language has explicit support for code sandwich patterns in the form of environments [15]. From the user's perspective, there are two commands of importance: `\begin{envname}` and `\end{envname}`. The user specifies a new environment by passing the name of that environment to `begin`, and closes the environment by passing the same name to

`end`. These operations correspond to pushing and popping from an environment stack. Matched behavior is enforced: the user may only call `end` with the name of the current top of the environment stack, or the input will be rejected as invalid.

Thus  $\text{\LaTeX}$  gives a guarantee of inevitability at “run time” (i.e., document generation time). The programmer who writes a `begin` clause may be certain that the corresponding `end` will be reached if the document is valid. However, `end` must be called explicitly, with a particular argument. This makes the  $\text{\LaTeX}$  language unique among the languages reviewed in this section: although *after* code must be called explicitly by the programmer, inevitability is still ensured.

## 5 Comparison of Existing Techniques

We now consider similarities and differences in code sandwich support for the languages discussed above.

### 5.1 Common Themes

The following points are common to all the approaches reviewed in Section 4.

- Of the three features discussed at the beginning of Section 4, inevitability recurs in every instance. This feature is clearly in high demand, particularly in languages supporting exceptions.
- Code sandwiches tend to have well-defined lifetimes: most code sandwiches require that *after* code run at a deterministic time. Each of the languages and extensions in our review provides this feature.
- When several code sandwiches are nested, they are organized into stacks. In most languages, they are associated with the call stack or with lexical scoping. Weimer [24] argues that this association may be too limiting, and his proposal separates compensation stacks from the lexical stack.

### 5.2 Syntactic Linkage and Encapsulation

Explicitly associating *before* and *after* code can be useful for both the programmer and the compiler: each benefits from the unambiguous connection between the two code segments. In an object-oriented language, it is natural to connect related functions or operations by grouping them and their shared state into a single class. This approach produces patterns such as C++ RAII objects and Python context managers. These objects provide the additional benefit of reusability: with the *before* and *after* code encapsulated into a single, reusable object, the programmer is saved the work of writing these code segments repeatedly.

Encapsulation within objects comes at the price of defining and allocating the objects. One criticism of the RAII idiom is that new objects must be defined and created whenever an inevitable *after* segment is required. This can be especially problematic in C++, where classes may only be defined at global scope. The compensation stack



approach provides syntactic linkage without encapsulation, and the D scope guard syntax is similar. In both, *before* and *after* are adjacent to one another, making their association clear (at least to the programmer). But they remain statements, not functions or classes, and thus do not form reusable sandwich encapsulations.

Table 1 summarizes the common code sandwich idioms that were presented in Section 4.

### 5.3 Sharing State

*Before* and *after* code segments typically manipulate the same memory. In our running mutex examples above, this shared state consists of the lock object. The means by which *before* and *after* share state affects how easily programmers can use a language’s code sandwich mechanisms.

When *before* and *after* are two methods of a single object, it is natural to share state using data fields in that object. We find this approach used with RAII (in C++) and with context managers (in Python and C#). When *before* and *after* are functions that do not share a single object’s context, then state sharing must be done either by global variables (as in L<sup>A</sup>T<sub>E</sub>X) or by matching arguments (as in Lisp). When *before* and *after* appear as immediate statements in a shared lexical context, as in Java or D, then the state sharing mechanism is the variables in that shared context. Such a shared context also exists between the two halves of a Python generator function or a Java closure.

Table 2 summarizes the state sharing mechanisms in the languages reviewed above.

### 5.4 Exceptions

One of the chief functions of explicit code sandwich support is to ensure that *after* code runs if an exception propagates out of the *body* code. Language designers must also address the possibility of exceptions in *before* and *after* code.

#### 5.4.1 Exceptions in *before*

A language must specify what happens if an exception escapes a *before* segment. Does the associated *after* code run? If so, then it may become necessary for the *after* code to explicitly check whether the *before* code succeeded, as in Figure 1. In many languages, *after* code is specified to only run if *before* code succeeds. Thus, in C++, an object only becomes eligible for destruction after its constructor has completed. However, we have noticed that many C++ programmers are unsure about this behavior.

#### 5.4.2 Exceptions in *after*

The case of exceptions thrown during cleanup actions can be tricky. Languages typically allow only one exception to be propagating at any time. An *after* segment may run in response to an exception, and yet it may encounter an exception itself. For example, an *after* segment that attempts to flush a file buffer and close the file may itself cause an exception if I/O fails.

Table 1: Variation of code sandwich solutions: how *before* and *after* code are typically defined, and the programming idiom that they represent.

Language/Extension	<i>before</i>	<i>after</i>	Idiom
C++ / RAII	constructor	destructor	2 class methods
Java, Python	try	finally	immediate in code
D Scope Guards	inlined	scope (...)	immediate in code
C# using statement	constructor	IDisposable.Dispose	2 class methods
Python CM – basic	__enter__	__exit__	2 class methods
Python CM – generator	before yield	after yield	1 function
Common Lisp	inline	argument to unwind-protect	immediate + 1 function
O’Caml		arguments to doWithCleanup	2 functions or closures
Scheme		arguments to dynamic-wind	2 closures
Compensation Stacks	compensate	with	immediate in code
Java with Closures	try	finally	body as closure argument
C cleanup chains	inlined	make_cleanup(...)	immediate + 1 function
L <sup>A</sup> T <sub>E</sub> X	begin	end	2 global functions

Table 2: Variation of code sandwich solutions: how *before* and *after* code share state.

Language/Extension	Sharing Mechanism
C++ / RAII	Fields of shared class
<code>try/finally</code>	Scope of local context
D scope guards	Scope of local context
C# <code>using</code> statement	Fields of shared class
Python CM – basic	Fields of shared class
Python CM – generator	Scope of shared function
Common Lisp	Arguments passed to <i>after</i>
O’Caml	Arguments passed to <i>before</i> , <i>after</i>
Scheme	Arguments passed to <i>before</i> , <i>after</i>
Compensation Stacks	Scope of local context
Java with Closures	Scope of shared function
C cleanup chains	One argument passed to <i>after</i>
L <sup>A</sup> T <sub>E</sub> X <code>begin/end</code>	Global variables only

C++ aborts a program that throws an exception from a destructor if that destructor ran due to exception-triggered stack unwinding. In Java, a `finally` block may throw an exception; any exception that was previously propagating is “swallowed” and its information lost. Python’s context managers also swallow exceptions if a new exception arises in a context manager’s `__exit__` function.

A related issue is whether or not an *after* action can determine whether it is executing because of an exception or because of normal execution. C++ provides a library function `uncaught_exception` that dynamically checks whether an exception is propagating. We are aware of no such mechanism in Java. The D `scope(failure)` (and `scope(success)`) guards allow cleanup actions to be invoked only in exceptional (or normal) executions.

## 5.5 Style

Style is subjective, so comparing the style of languages is challenging. However, there are several concrete stylistic choices that must be made when adding explicit code sandwich support to a language.

**Adjacency** Do *before* and *after* code appear next to each other in the source, or are they separated by the *body*?

**Locality** Do the *before* and *after* code appear next to the *body* code, or are they in a separate place?

**Verbosity** Is code sandwich support easy for programmers to use, reuse, and understand?

Table 3: Variation of code sandwich solutions: adjacency (of *before* and *after* code) and locality (of *before* and *after* with respect to *body* code). See Section 5.5. Data in this table are based on the usage patterns presented in Section 4.

Language/Extension	Adjacency	Locality
C++ / RAII	Potentially adjacent	not local
<code>try/finally</code>	Not adjacent	surrounds <i>body</i>
D scope guards	Potentially adjacent	above <i>body</i>
C# <code>using</code> statement	Potentially adjacent	not local
Python CM – basic	Potentially adjacent	not local
Python CM – generator	Always adjacent	not local
Common Lisp	Not adjacent	surrounds <i>body</i>
O’Caml	Potentially adjacent	not local
Scheme	Not adjacent	surrounds <i>body</i>
Compensation Stacks	Always adjacent	above <i>body</i>
Java with Closures	Potentially adjacent	not local
C cleanup chains	Potentially adjacent	above <i>body</i>
LaTeX <code>begin/end</code>	Not adjacent	surrounds <i>body</i>

Table 3 considers each of our example languages in terms of adjacency and locality. We have listed destructors and context managers as “potentially adjacent” because it is up to the programmer whether the *before* and *after* functions appear next to each other in the code. We characterize certain solutions “not local” when their *before* and *after* segments are not given explicitly in the context of the *body*. This is not necessarily a bad thing, since whenever the programmer must do something explicitly, he has the opportunity to forget to do it.

Verbosity is difficult to quantify. In Section 7, we show Java examples where complete exception safety requires deeply nested `try/finally` constructs. Other solutions, such as C#, provide syntactic sugar to help limit nesting depth (see Section 4.4).

Several commentators, including Boehm [2], have raised a stylistic complaint against C++ destructors: that a closing brace may execute arbitrary code via destructor invocation. Many of the other designs we review above are subject to the same criticism. In the limit, it would be possible to use a feature like D’s scope guarded statements to write an entire function in reverse order of its actual execution. As with most matters of style, there is no universally accepted solution. All language features can be misused, and good taste is not a statically-enforceable property.

## 6 Jyro: Explicit Support in Java

A language’s code sandwich constructs influence how programmers structure, write, and reason about their code. To demonstrate this difference we implement Jyro, a small extension to Java, and provide examples where code sandwiches are significantly easier

to read and write in Jyro than in Java.

## 6.1 Proposed Syntax and Semantics

Jyro is built atop JastAddJ [7], an extensible Java compiler. Jyro adds the keyword `within`, which introduces a syntactic block that uses encapsulated code sandwiches. The syntax for `within` is similar to C#'s `using` syntax. For example, consider the use of a critical region class designed for Jyro:

```
// L is a mutex lock
within (new Locked( L )) {
    /* body code */
}
```

The class `Locked` in this code must implement one of two interfaces: `Sandwich` (proposed here) or `java.io.Closeable` (already in standard Java). The `Sandwich` interface consists of two public methods:

- `void before();`
- `void after();`

The Jyro compiler introduces a hidden variable `tmp` to hold the value of the expression in `within`'s parentheses. If `Locked` implements `Sandwich`, then Jyro compiles the above `within` block as:

```
{
    final Locked tmp = new Locked( L );
    tmp.before();
    try {
        /* body code */
    } finally {
        tmp.after();
    }
}
```

The standard `java.io.Closeable` interface provides a `void close()` method. Many classes in `java.io` implement `Closeable`; all such classes expect a call to `close` to postdominate their constructor. If `Locked` implements `Closeable` and not `Sandwich`, then Jyro compiles our example as:

```
{
    final Locked tmp = new Locked( L );
    try {
        /* body code */
    } finally {
        tmp.close();
    }
}
```

The guard of `within` may be any expression, not just an allocation. Optionally, the guard may introduce a named variable to hold the `Sandwich` or `Closeable` object, so the *body* can access other the fields and methods of that object. Thus:

```
within (Locked crit = createLock( L )) {
    /* body code */
}
```

Jyro compiles this as the following:

```
{
    final Locked tmp = createLock( L );
    Locked crit = tmp;
    tmp.before();
    try {
        /* body code */
    } finally {
        tmp.after();
    }
}
```

Observe that any assignment to `crit` within the *body* will not change the object on which `after` is called.

## 6.2 Classification by Features

Regarding the desiderata of Section 5, the `within` construct provides **inevitability** with a well-defined lifetime tied to the lexical stack. A class that implements `Sandwich` has explicit **syntactic linkage** of *before* code with *after* code. Classes that implement `Closeable` have weaker linkage; the `close` method is generally meant to postdominate the constructor. In either case, **encapsulation** is strong as *before* and *after* code are bound in a single class from which many instances can be created.

Fields in the `Sandwich`- or `Closeable`-implementing object allow **state sharing** between *before* and *after* code, much like C++ or C#. **Exceptions in before code** propagate to the caller without running *after* code. **Exceptions in after code** also propagate to the caller, but will swallow any exception that might already have been propagating up from the *body*. This is similar to exception-swallowing in Python and standard Java.

Jyro allows **adjacency** in definitions of *before* and *after* code. Jyro does not provide **locality**: an encapsulated sandwich class is typically defined far away from the *body* code. **Verbosity** is good: Jyro is syntactically concise, especially when a single `Sandwich` or `Closeable` implementation is reused in many places. Section 7.1 shows that Jyro would eliminate thousands of lines from a large, complex, real-world application.

Of the languages reviewed in Section 4, Jyro is most similar to C#. The chief difference is that Jyro's `Sandwich` interface specifies two functions, representing both *before* and *after* code, in contrast to C#'s `Dispose` method, which only represents

*after* code. Our goal in proposing Jyro is to explore the design space of code sandwich support, not to assert the superiority of one approach over another.

## 7 Jyro Case Studies

To assess the utility of a structure like `within`, we reworked portions of two large, complex Java applications to use Jyro.

Azureus is a popular open-source BitTorrent client written in about 340,000 lines of Java. Azureus often handles network connections and files, and its authors have clearly attempted to handle the implicit code sandwiches associated with managing these resources. These attempts often cause confusing, deeply-nested `try/finally` blocks. Despite these efforts, numerous code sandwich bugs remain. Jyro's `within` syntax both simplifies confusing sections and makes many current bugs easy to correct.

Lucene [1] is a high-performance open-source library for text search engines. Lucene is about ten thousand lines long and is a fairly mature project; we found few code sandwich bugs. The sandwich bug analysis of Weimer and Necula [25] detected only two possible bugs, one in a test case and one reachable only if a core Java library was buggy and several exceptions occurred. However, we found several places where Jyro's `within` syntax makes Lucene code significantly easier to read and understand.

### 7.1 Monitor Class

In Azureus the class `AEMonitor` implements thread-queuing synchronization. Azureus contains over 1100 calls to `AEMonitor.enter`. Each is protected by a `finally` block containing just a call to `AEMonitor.exit`. These calls often surround only one or two statements, but Azureus contains cases where matching `enter` and `exit` calls surround over 200 lines.

It is clear from usage that these calls are the *before* and *after* of a code sandwich, although this is nowhere documented in Azureus. Still, this was handled correctly in every case we examined, but at a mild cost to code complexity:

```
mon.enter();
try {
    /* body code */
} finally {
    mon.exit();
}
```

In Jyro, it is trivial to rewrite `AEMonitor` as a class that implements `Sandwich`. Every use of the class is then simplified:

```
within (mon) {
    /* body code */
}
```

Rewriting uses of `AEMonitor` in this manner eliminates over 3000 lines of Azureus code and guarantees that correct sandwiches will remain correct in the future.

## 7.2 File Stream Bugs

Azureus handles many files and network streams, so they are frequently opened and closed throughout its code. Azureus's authors clearly try to close every file they open, even when exceptions are thrown, but mildly complicated situations often confound their efforts.

For example, the following code (heavily condensed here for space) appears within the `AEJarReader` class:

```
1  InputStream is = null;
2  JarInputStream jis = null;
3  try {
4      is = getStream(name);
5      jis = new JarInputStream(is);
6      /* ... read streams ... */
7  } catch (Throwable e) {
8      Debug.printStackTrace( e );
9  } finally {
10     try {
11         if ( jis != null ){
12             jis.close();
13         }
14         if (is != null){
15             is.close();
16         }
17     } catch (Throwable e) {
18     }
19 }
```

The bug in the above code is easily overlooked, and is found throughout Azureus. If `jis.close` throws an exception on line 12, its surrounding try block will be aborted and `is.close` on line 15 will not execute. We can correct this problem as follows in plain Java:

```
1  try {
2      try {
3          InputStream is = getStream(name);
4          try {
5              JarInputStream jis =
6                  new JarInputStream(is);
7              /* ... read streams ... */
8          } finally {
9              jis.close();
10         }
11     } finally {
12         if (is != null) {
13             is.close();
14         }
15     }
16 }
```



```

14         }
15     }
16 } catch (Throwable e) {
17     Debug.printStackTrace( e );
18 }

```

The call to `is.close` on line 13 requires a guard because its constructing function may return null on line 3. The call to `jis.close` on line 9 needs no guard because the new object allocation on line 5 cannot return null.

The corrected code above is so complicated that the author may have intentionally ignored the bug for the sake of clarity. Using Jyro yields code that is not only correct but also easier to read, write, and think about:

```

try {
    InputStream is = getStream(name);
    if (is != null) {
        within (is) {
            within (JarInputStream jis
                = new JarInputStream(is)) {
                /* ... read streams ... */
            }
        }
    }
} catch (Throwable e) {
    Debug.printStackTrace( e );
}

```

This type of bug appears repeatedly in Azureus, even though the authors clearly intend to satisfy code sandwich requirements when exceptions are thrown. These bugs indicate that either programmers have difficulty reasoning about exceptions or that programmers find the `try/catch/finally` statement to be specifically confusing. In either case, the `within` statement makes it easier to get such code right.

### 7.3 Sandwich Nesting

Now consider a more extreme (but not artificially-constructed) example where code in Jyro is preferable to the same code in Java. Consider first a trimmed section of code from Azureus's `CorePatcherChecker` class:

```

try {
    InputStream is =
        new FileInputStream("Azureus2.jar");
    InputStream pis =
        new FileInputStream(/*filename*/);
    OutputStream os =
        new FileOutputStream("t.jar");
    new JarPatcher(is, pis, os);
}

```

```

        is.close();
        pis.close();
        os.close();
    } catch (Throwable e) {
        Debug.printStackTrace( e );
    }
}

```

This code is rife with code sandwich defects. We first rewrite the code in pure Java using only try/finally:

```

try {
    InputStream is =
        new FileInputStream("Azureus2.jar");
    try {
        InputStream pis =
            new FileInputStream(/*filename*/);
        try {
            OutputStream os =
                new FileOutputStream("t.jar");
            try {
                new JarPatcher(is, pis, os);
            } finally {
                is.close();
            }
        } finally {
            pis.close();
        }
    } finally {
        os.close();
    }
} catch (Throwable e) {
    Debug.printStackTrace( e );
}

```

Contrast with a Jyro version that is shorter than even the original, buggy Java code:

```

try {
    within (InputStream is =
        new FileInputStream("Azureus2.jar"))
    within (InputStream pis =
        new FileInputStream(/*filename*/))
    within (OutputStream os =
        new FileOutputStream("t.jar")) {
        new JarPatcher(is, pis, os);
    }
} catch (Throwable e) {
    Debug.printStackTrace( e );
}

```

## 7.4 Wrapper Reuse

Lucene contains classes designed to act as Sandwich objects act in Jyro. Pure Java makes this usage cumbersome. The abstract inner class `Lock.With` in Lucene contains two key functions, the abstract method `doBody`, and the method `run`. A simplified excerpt from Lucene demonstrates its normal usage of this class:

```
new Lock.With(lock, timeout) {
    public Object doBody()
        throws IOException {
        /* body code */
    }
}.run();
```

That is, “make a dummy instance of the abstract class `Lock.With` with the following block to implement `doBody`, and run this instance.” This pattern is similar to the access pattern for the Java privileged code API, discussed in Section 4.2.1. It appears six different times in the Lucene sources.

Jyro is built to handle exactly such cases. After a simple rewrite of the `Lock.With` class to implement the `Sandwich` interface, the above code reduces to:

```
within(new Lock.With(lock, timeout)) {
    /* body code */
}
```

This code is easier to understand, involves less boilerplate code, and is more comprehensible to programmers who may be uncomfortable with anonymous inner classes.

## 8 Conclusions

Prior work has demonstrated the existence and importance of code sandwich defects in real world programs. Programmers and language designers have developed varying conventions and language support that aid the creation and understanding of code sandwich patterns. In many cases, sandwich support is not designed into languages but rather repeatedly evolves as an idiomatic use of other primitives. The results may not be what language designers would have intended had they considered their options more directly. In our review of these approaches, we explore their similarities and differences, and we offer a common framework for comparing them.

Armed with our understanding of code sandwich patterns, we define and implement Jyro, a small extension to Java with language-level support for encapsulated, reusable code sandwiches. We assess Jyro’s utility by applying it to two large, mature, real-world Java programs, and find numerous examples of incorrect or unreadable code that can be fixed or improved. Future language designers would do well to make carefully-considered choices among the design alternatives we have identified.

## 9 Acknowledgments

This research was supported in part by AFOSR Grant FA9550-07-1-0210; NSF Grants CCF-0621487, CCF-0701957, and CNS-0720565; and an NDSEG Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of AFOSR, NSF, DoD, or other institutions.

## References

- [1] The Apache Software Foundation. Apache Lucene Project. <http://lucene.apache.org/java/docs/>, 2007.
- [2] Hans-Juergen Boehm. Destructors, finalizers, and synchronization. In *POPL*, pages 262–272, 2003.
- [3] Boost. Boost C++ Libraries. <http://www.boost.org>, March 2008.
- [4] Gilad Bracha, Neal Gafter, James Gosling, and Peter von der Ahé. Closures for the Java programming language (v0.5). <http://www.javac.info/closures-v05.html>, December 2007.
- [5] Digital Mars. D Programming Language. <http://www.digitalmars.com/d/>, 2008.
- [6] Ecma International. *Standard ECMA-334: C# Language Specification*. Ecma International, Geneva, fourth edition, June 2006.
- [7] Torbjörn Ekman and Görel Hedin. The JastAdd extensible Java compiler. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *OOPSLA*, pages 1–18. ACM, 2007. ISBN 978-1-59593-786-5.
- [8] Li Gong. Java™ security architecture (JDK1.2), version 1.0. Technical report, Sun Microsystems, October 1998.
- [9] Li Gong and Roland Schemers. Implementing protection domains in the Java™ Development Kit 1.2. In *NDSS*. The Internet Society, 1998. ISBN 1-891562-01-0.
- [10] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, London, third edition, 2005.
- [11] Richard Kelsey, William D. Clinger, and Jonathan Rees. Revised<sup>5</sup> report on the algorithmic language Scheme. *SIGPLAN Notices*, 33(9):26–76, 1998.
- [12] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

- [13] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In Jørgen Lindskov Knudsen, editor, *ECOOP*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, 2001. ISBN 3-540-42206-4.
- [14] Henry F. Korth, Eliezer Levy, and Abraham Silberschatz. A formal approach to recovery by compensating transactions. In Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek, editors, *VLDB*, pages 95–106. Morgan Kaufmann, 1990. ISBN 1-55860-149-X.
- [15] Leslie Lamport. *TeX: A Document Preparation System – User’s Guide and Reference Manual*. Addison-Wesley Professional, Indianapolis, Indiana, second edition, June 1994.
- [16] Gary McGraw and John Viega. Privileged code in Java: Why the API changed from JDK1.2beta3 to JDK1.2beta4. In *Developer.com Gamelan*. Jupitermedia, August 1998. <http://www.developer.com/java/other/article.php/604131>.
- [17] Alexander Nasonov. Boost.ScopeExit. [http://boost-consulting.com/vault/index.php?action=downloadfile&filename=scope\\_exit-0.04.tar.gz&directory=&](http://boost-consulting.com/vault/index.php?action=downloadfile&filename=scope_exit-0.04.tar.gz&directory=&), August 2007.
- [18] K. Pitman (ed). Common Lisp HyperSpec. <http://www.lispworks.com/reference/HyperSpec/>, 2001.
- [19] Darrell Reimer, Edith Schonberg, Kavitha Srinivas, Harini Srinivasan, Bowen Alpern, Robert D. Johnson, Aaron Kershenbaum, and Larry Koved. SABER: smart analysis based error reduction. In George S. Avrunin and Gregg Rothermel, editors, *ISSTA*, pages 243–251. ACM, 2004. ISBN 1-58113-820-2.
- [20] Guido van Rossum. *Python Reference Manual*. Python Software Foundation, September 2006. <http://docs.python.org/ref/ref.html>.
- [21] Guido van Rossum and Nick Coghlan. PEP-343: The “with” statement. <http://www.python.org/dev/peps/pep-0343/>, July 2006.
- [22] Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton van Straaten, Richard Kelsey, William Clinger, and Jonathan Rees. Revised<sup>6</sup> report on the algorithmic language Scheme. <http://www.r6rs.org/>, 2007.
- [23] Bjarne Stroustrup. *The design and evolution of C++*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994. ISBN 0-201-54330-3.
- [24] Westley Weimer. Exception-handling bugs in Java and a language extension to avoid them. In Christophe Dony, Jørgen Lindskov Knudsen, Alexander B. Romanovsky, and Anand Tripathi, editors, *Advanced Topics in Exception Handling Techniques*, volume 4119 of *Lecture Notes in Computer Science*, pages 22–41. Springer, 2006. ISBN 3-540-37443-4.

- [25] Westley Weimer and George C. Necula. Finding and preventing run-time error handling mistakes. In John M. Vlissides and Douglas C. Schmidt, editors, *OOPSLA*, pages 419–431. ACM, 2004. ISBN 1-58113-831-8.