Introducing Mutable Environments

Philip Machanick
School of ITEE, University of Queensland
Brisbane, Qld 4072, Australia
philip@itee.uq.edu.au

Abstract

Mutable environments are a proposed addition to object-oriented languages, in which the *environment* in which a method runs can be changed from the defaults specified in its class. For this purpose, *mutable* methods are defined, some of which are predefined and are invoked at specific times. Others can be added by a programmer. A mutable method can be overridden by one or more *environment objects*, in whose scope a method runs. Mutable environments allow concerns for which aspect-oriented programming was developed to be addressed in a relatively simple framework, which is general enough to support wrappers around methods and error handling. This paper presents a first approximation to the idea, with some preliminary evaluation.

1 Introduction

Mutable environments are a proposed extension of the ideas of object-oriented programming to allow more general alterations than derived classes to previously-defined program components. Object-oriented programming, though a paradigm with growing acceptance by comparison with its early decades, has well-known weaknesses. Reusing components, while a powerful concept and well-supported in some languages, has not always met with success in the real world, and there are reports of resistance to reuse-based strategies in real projects [3]. There are long-standing concerns that learning object-oriented concepts is not as easy at it should be [1, 5, 6]. An inheritance hierarchy has also been identified as being unsuited to representing cross-cutting abstractions, such as error-handling and synchronization.

Addressing these concerns requires that aspects of programming which do not naturally fit an inheritance hierarchy be decoupled from the hierarchy. To do so would address the concern of cross-cutting abstractions being ill-suited to a strict hierarchy; addressing the other concerns is more difficult to evaluate. These concerns could, for example, be a consequence of teaching technique [13]. However, allowing programming approaches outside the inheritance hierarchy could make for a more natural style in some cases.

The issue of cross-cutting abstractions has been addressed in aspect-oriented programming [10] and in Timor's bracket methods [8]. This paper argues for a simpler mechanism for changing component behaviours, called mutable environments, which makes the environment in which a method executes explicit and capable of being changed. A class can have methods which are defined to be `mutable`, and an environment enclosing a method call may redefine an `mutable` methods used in that method. A key difference from the Timor mechanism is that the environment class need not be specifically designed as such but the mutable class needs to be defined as mutable. Timor does the opposite – the qualified type is not defined in any special way, but the type which qualifies it is specifically designed as such [9].

The mechanism is to initialize an object which has the lifetime of one or more method invocations, the *`Environment`*. The `Environment` is initialized like any other object, but has methods specific to processing before and after a method it encloses. Like any other object, it can be replaced by another descended from it, with different behaviours. Further, it can be copied, allowing a new invocation to be initialized with a saved `Environment`. A variation on the same idea can be used in error-handling. Another kind of mutable environment, derived from class `ErrorHandler`, replaces a mutable method `catch` in a class. If the `ErrorHandler` environment defines a `catch` method, it replaces the default method in the class.

This paper explores how this relatively simple additional feature – an environment which replaces a mutable method – can support flexibility in design and change of object-oriented programs. In the spirit of exploring a new concept rather than inventing yet another language, the concept is explored in the abstract (using a simple object-oriented pseudocode), rather than as a construct in a specific language. However, some ground rules are assumed. Instance variables (member variables for the C++ inclined) are always hidden inside a class, and interfaces and implementations are separate, allowing alternative implementations. Otherwise, assumptions are minimized (e.g., whether there is or is not garbage collection is not central to the model; whether errors are propagated or cause a crash if not handled immediately is also not explored). Even the assumptions which have been made are to simplify the examples, and are not central to the ideas of the paper.

A specific design objective is to provide a mechanism to implement cross-cutting abstractions which are orthog-

onal to a class hierarchy with as few changes from conventional object-oriented languages as possible. The aim is to achieve the goals of aspect-oriented programming as simply as possible to facilitate ease of implementation, and ease of learning the concepts.

The remainder of this paper is organized as follows. Section 2 provides some background, including a survey of similar ideas. Section 3 outlines the idea in more detail, with illustrative examples, and Section 4 provides a preliminary evaluation of the idea, with some thoughts on implementation. Finally, conclusions wrap up the paper in Section 5.

2 Background and Related Work

Languages have a variety of mechanisms to address cross-cutting concerns such as synchronization and error-handling. Some languages build these features in, while others rely on library and (also popularly called application programming interface – API) designers to provide the features.

Java, for example, provides synchronization as a language feature. In doing so, it provides a special feature, rather than a general mechanism, which could be adapted to other cross-cutting concerns. Further, Java's virtual machine approach means that synchronization can in principle be implemented efficiently for its platform once, though implementation on a specific machine may still take work. Still, building a mechanism into a language limits flexibility, and places programmers at the mercy of developers of the underlying virtual machine, which may not always be efficient [2]. In C++, there is no standard for synchronization. While many libraries exist, users are left with a range of choices [15], which may not be mutually compatible, creating problems with mixing libraries. Designers of specific APIs need to define strategies of their own, which run into the problem of the difficulty of fitting in constructs which cut across the class hierarchy.

Against this background, Aspect-Oriented Programming (AOP) attempts to provide mechanisms to implement abstractions across a class hierarchy [10].Work in this field has included pattern languages for defining crosscuts [7], and exception-handling in aspect-oriented programming [12].

An idea similar to that of this paper is *context relations*, a strategy for patching composite objects together. Context relations are somewhat more complicated than the mechanism defined here, and also emphasize composing a new class out of a specific collection of classes [14]. The approach used here is more flexibile in object composition, which can be on a per-method basis (either in a class definition, or at method call).

An approach related to AOP is bracket methods in qualified types in Timor [9, 8]. The approach in Timor is one of qualifying types, which is a little more restrictive than the approach developed here, which can encapsulate a method call in a new environment at any call. Timor's mechanisms, while simpler than aspect-oriented programming, are more complicated than those proposed here. Further, the approach in Timor is to specify types (classes) without indicating that they can be modified, whereas a qualifying type (which plays the role of an environment class here) has to be specifically defined as such. The approach here requires that the class being modified define methods as mutable, whereas environment classes need not have any special features. This approach seems more natural in terms of security – a class which is being modified needs to be defined to permit this explicitly. Even if the class which modifies it was not designed originally for this purpose, its abstraction secrets remain hidden because the mutable class using it has to match its public interface. Further, the mutable environments approach allows the option of using classes not designed initially as environment classes in this role, which has the potential for increasing design flexibility.

In languages with weaker type checking such as Smalltalk, it is easier to implement orthogonal concepts (e.g., exception handlers [4]), since type checking doesn't interfere with mixing components from different parts of the class hierarchy.

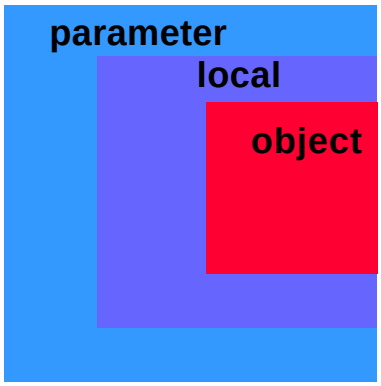
In general, component-based approaches are inherently fragile under evolutionary change, because they have in them the implicit assumption that interfaces and behaviours can be precisely specified [11]. Mutable programming has the potential for designing systems with evolution in mind – mutable methods can be used where change is considered likely. Even where mutable methods are not used in the initial design, change can be incorporated by making methods mutable – a relatively simple design and implementation change – and redefining them as needed.

Mutable environments are an attempt at rolling a variety of mechanisms into one relatively simple approach at providing a layer of abstraction across the class hierarchy. The intent is to arrive as close as possible to the flexibility of a non-typed language, without giving up typing. If special mechanisms in other languages can be implemented with a minimal, general feature set, other, more general options can be explored.

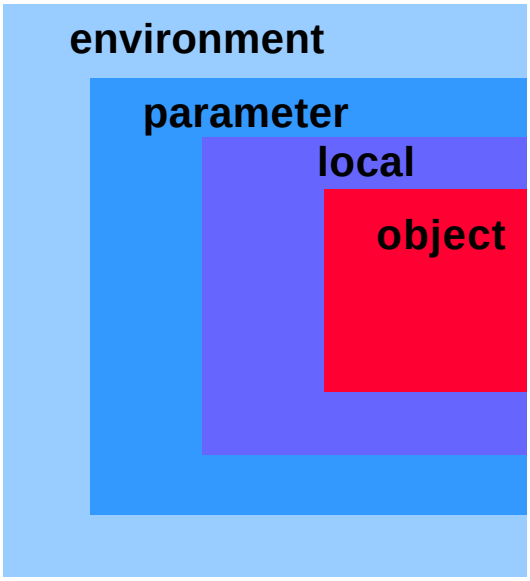
3 Essentials of Mutable Environments

In a conventional object model during a method invocation, objects are conceptualized as in fig. 1(a). The idea introduced here is to see an object as consisting of the object as defined by its class (and any operations since creation) and the environment of the current method call, with additional constraints and behaviours as in fig. 1(b).

At first sight, fig. 1(b) would appear to simply be a picture illustrating what happens in a conventional object-oriented model. An object exists in an environment of globally-defined names. However, the difference proposed in the mutable environments model is that the environment is constructed as an object, encapsulating the object which



(a) conventional view of data in an object



(b) composite object including environment

Figure 1: Objects viewed conventionallly and as proposed here.

called the method, introducing methods which can be defined as part of the environment. Further, methods in a class can be defined as `mutable`, meaning that they can be overridden by an enclosing environment. A top-level class, `Object`, is assumed, with a standard set of mutable methods corresponding to starting a new call, and terminating a call.

The remainder of this section outlines standard methods which would be part of any environment, and goes on to outline a mechanism for overruling standard behaviours, which can apply to situations other than environments. A model of defining environments as classes which can be separately created, copied and applied to a method is detailed. Error handling is then outlined, followed by a summary.

3.1 Standard Environment Methods

An environment needs some minimal standard methods to provide the behaviours normally associated with environment creation and destruction, as well as methods to support other useful standard behaviours. Further, it is useful to have defaults, so the absence of a an explicit environment means that method invocation works as before.

Assume 3 top-level classes: `Object`, `Environment` and `ErrorHandler`. Class `Environment` defines methods `initial` (a constructor), `final` (called on object destruction – provided for compatibility with languages with explicit memory deallocation), `start` (called before a method it encloses starts execution), and `exit` (called after a method it encloses returns). `ErrorHandler` has a constructor and finalizer which in many cases will be placeholders, and defines one method for general use, `catch`. `Object` also defines methods for `start`, `exit` and `catch`, but defines them as `mutable`, meaning they can be replaced by environment methods. Defining a method as `mutable` is done by adding a `mutable` keyword. For example, in the top-level class definition – assuming a a type `Action` which has values `terminate`, `continue` and `propagate`:

```
class Object
  initial (); // constructor
  final ()  // called on object deallocation
  // called when any method starts:
  mutable start ();
  mutable exit (Action kill, bool returnsVal,
    Object returnThis);
  // doesn't return, invokes exit:
  mutable catch (Error errorCondition);
endc;
```

The `catch` method, if called, by default returns control to the `exit` method rather than the place it was called. It is possible to define a fine-grained error handler mechanism by overloading `catch`, and allowing multiple definitions for different parameters. In such a model, the most accurate match would have to be found. For purposes of developing the concepts, though, the class mutable version of `catch` is only called if it is not defined in an `ErrorHandler` environment (i.e., there is for practical purposes only one version of `catch` in a given context). An issue which needs to be dealt with is recoverable errors – where the error handler may need to take some action, but the overall effect is that the method returns. There is no difficulty if the method doesn't return a value but if it does, it is necessary to have a way of returning the value. The approach adopted here is to construct the `Error` object as including a reference to the return value (as constructed so far), with an instance variable indicating whether a return value is included. If the `Error` object was given such a reference, the error handler would return it, otherwise it would return without returning a value.

`ErrorHandler` needs to have a constructor taking a reference to a result, which could be `null` if there is no result to return (alternatively, it could have a second con-

structor with no parameters in a language supporting overloading).

An example using error handling (to jump ahead a bit – the notation is defined in 3.2) to read to end of file in a method which returns the values read in a container called `result` could look like this:

```
in new ErrorHandler (result)
  while (true)
    result.add (system.in ());
```

The default methods of `Environment` take no parameters, except for `exit`, which takes parameters indicating whether it has been called to kill the program and if not, whether it should return a result and if so, where to find the result. The constructor can be redefined in a derived class as having parameters, but the other methods can't have parameters redefined because they are not called explicitly. By default, the methods do nothing, except. the default `catch` (if `ErrorHandler` is not redefined) exits the program. When an error is detected, an `Error` object is created, and `catch` is called with a `Error` object as a parameter. If the class where the error was detected has not defined its own (non-mutable) `catch` method, and the environment has a `catch` method, this method is called. Otherwise the method is exited. Options for further propagating an error in are not essential to the ideas here and are not explored in this initial discussion of mutable environments. Varying error handling is therefore a property of the specific class of the `Error` object and the `catch` method either defined in the class, or in the environment.

3.2 Definition of Environment Classes

Defining an environment as a class has potential benefits, such as allowing an environment definition to be specialized, using inheritance. The approach explored here is one of defining an *environment* class which extends the environment of a method, much the way bracket methods do in Timor. In the spirit of providing a basis for comparison with Timor, examples used to build the ideas are similar to the examples used to develop the ideas in Timor. However, given the differences in the approach, the order of presentation of concepts differs from the Timor description [9].

An environment needs to define the standard methods given in 3.1, if any of these need to be modified. As an example, consider a synchronized environment, using a simple lock to protect access. This environment needs to encapsulate a lock variable (identifying which lock it corresponds to). It needs to acquire the lock on entry to the method in its environment, and release the lock on exit. Since the lock can potentially be held by more then one object, acquiring and releasing the lock are best done in the `start` and `exit` methods. Initializing and deallocating the lock (assuming no garbage collection, in the presence of deallocation would disappear) would be done respectively in the `initial` and `final` methods. Overall, the class could look like this (assuming a notation in which interfaces and implementations are separate):

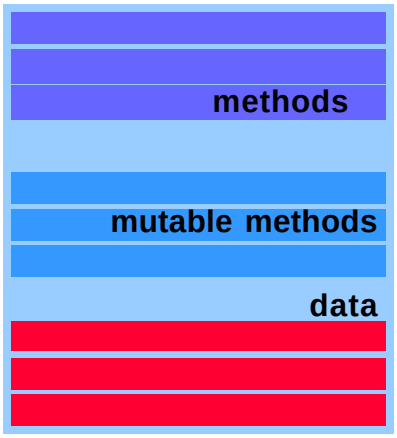
```
class interface Synchronized
    extends Environment
  initial ();
  start ();
  exit (Action kill, bool returnsVal,
    Object returnThis);
  final ();
endc;
class implementation Synchronized
  initial ();
    lockvalue = Lock.new ();
  endm;
  start ()
    lockvalue.acquire ();
  endm;
  exit (Action kill, bool returnsVal,
      Object returnThis)
    lockvalue.release ();
    parent (kill, returnsVal, returnThis);
  endm;
  final ()
    lockvalue.deallocate ();
  endm;
  data
    Lock lockvalue;
endc;
```

Use of such a class could take various forms. In the interests of exploring the design space, three alternatives are proposed here: ad hoc patching of calls, defining a specific behaviour as part of a class and defining a specific behaviour as part of creating a new implementation of a class. The first two approaches are explored in detail; the last is dependent on having a language in which implementations are separate, so it is discussed in less detail.

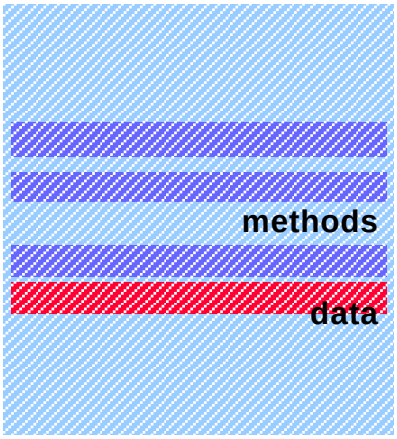
In all cases, the effect while a method is in progress is as illustrated in fig. 2. While the method is executing, it is in a composite object (fig. 2(c)), composed by merging the method's class (fig 2(a)) with the environment (2(b)). In the composite class, the mutable methods in the original class may sometimes (not always) be replaced by the environment, and the environment may introduce its own methods and data. However, since the environment is not (in general) known to the mutable class when it is defined, any new methods or data it introduces are for its internal use.

In all examples in this paper, only instance methods are shown as mutable, but the principle can apply to class methods, as long as they are class methods in both the mutable and the environment classes. Nothing is said about public and private methods in the examples. An important principle is that using a class as an environment should not expose its hidden features. However, there is no reason that a mutable class should not define mutable methods as private, even though they use public methods in an environment class.

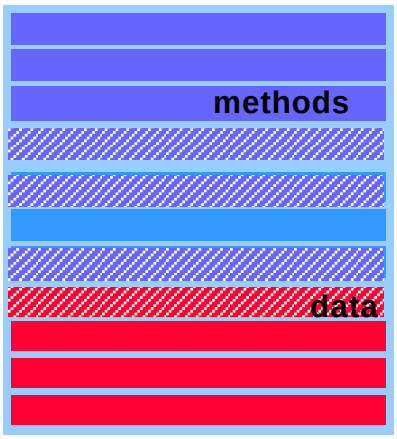
The remainder of this subsection presents detail of patching calls and adding environments to classes.



(a) mutable class



(b) environment class



(c) merged class

Figure 2: The effect of running a method in an environment.

Ad hoc patching of calls

To wrap a call in an environment, the notation proposed here is to precede the call by the keyword `in`, followed by the object containing the environment, and finally the call. In the simplest case, an object of the environment class is created directly as part of the call, and is discarded afterwards. It need not be given a name, but appears as creation of a new instance of an environment class. This notation has already been seen in the example of an error handler in 3.1.

The semantics of attaching an environment in this way to a method call are that the environment is initialized, parameters to the method are evaluated and bound, the `start` method of the environment is called, local variables are evaluated and bound, the method body is executed, local variables are unbound and deallocated, the `exit` method of the environment is called and finally, the `final` method of the environment is called. In the event of an error, the `exit` call would be invoked earlier; details are presented later.

Implementing a synchronized version of an abstract method invocation now becomes very simple, as in this example:

```
in new Synchronized ()
  object.method  (parameter);
```

This example needs further work, because we have no way of having another call use the same lock, which is a motivation to move on to the mechanism for saving and reusing an environment. In a situation where only one lock was needed, the lock variable could be a class variable in the `Synchronized` class. However, for generality, it would be preferable to allow more than one lock variable. Since the environment object is anonymous and no reference to it is available, a language without garbage collection would have to deallocate it on exit from the method.

That introduces the next mechanism: initializing a variable of an `Environment` class, and enclosing the default environment in this object (as opposed to the previous example, where an anonymous object enclosed the default environment). Let's create an object of the `Synchronized` environment class, and demonstrate the notation for running a method in that environment:

```
Environment locked = new Synchronized ();
in locked
  object.method (parameter);
```

If anyone else needs to use the same lock, they need to do a similar method call to that in this last example, except now the first line would fall away since the environment object already exists. If another lock is needed, creating a new instance is done as in the first line of the example. In a language without garbage collection, the environment object would be explicitly deallocated, in the same way as any other object.

Adding Environments to Classes

This kind of ad hoc patching could be useful in many cases. However, it may sometimes be better to patch a method once, at the place it's defined, rather than everywhere it's called. For example, if a class is defined in which a specific method, inherited from a parent class, now needs to run in a new environment consistently, it would be useful to be able to specify an environment in the class definition.

First, let's look at defining an environment as part of a class, then look at defining one as part of a specific implementation.

Assuming the existing Syncrhonized environment class, let us define a Buffer class, which implements bounded-buffer readers and writers. In the class interface, we do not specify a variable to contain the lock so that the implementation can be kept general[1]: it can either use a variable or a class name, resulting in the two different cases previously explored. The implementation in this example introduces a variable, to allow the reader and writer in the same buffer to share a lock unique to their buffer. Note that we have *not* specified a constructor for the environment in the class interface, yet, in the implementation, there is a a separate definition of the constructor for the environment. We will see in the next example why this is a useful notation.

```
class interface  SynchBuffer extends Buffer
  // constructor for object of this class
  initial ();
  in Synchronized Object read ();
  in Synchronized write (Object newData);
endc;
class implementation SynchBuffer
  initial ()
    parent ();
  endm;
  // constructor for environment
  in Synchronized lock initial ()
    lock = new  Synchronized ();
  endm;
  // final method deallocates environment
  // if no garbage collection:
  final ()
    deallocate lock;
  endm
  in lock Object read ()
    return parent ();
  endm;
  in lock write (Object newData)
    parent (newData);
  endm;
endc;
```

Note that an instance variable for the environment is implicitly declared in the constructor for the environment. If more than one named environment is used (for different methods) there would be a constructor of the relevant environment class for each new named environment. The keyword "parent" is used to invoke the parent class method of the same name. A derived class of SynchBuffer can redefine its environment class by one derived from the original, or leave it unchanged. It can also add a new environment class, which is used as specified in any derived methods.

If we (for some reason which would make sense with a different example) wanted a new lock for each call, the notation in the implementation would change to

```
class implementation SynchBuffer
  in new Syncronized () Object read ()
    return parent ();
  endm;
  in new Syncronized () write (Object newData)
    parent (newData);
  endm;
```

The implicitly declared instance variable for the lock and its initialization in a constructor would no longer be needed. As with ad hoc method patching, in this case, the anonymous objects would be implicitly deallocated after each call.

In the last two examples, the read and write methods don't define anything new: they invoke the parent method, but in now an environment where a lock variable is defined.

Let us now consider redefining a class using inheritance, and changing the environment type. That brings us to an explanation of why the environment constructor is separate from the ordinary constructor.

This class, if redefined using inheritance, would retain the Synchronized environments. However, a derived class could replace the initial method by one which creates a different environment type. The problem with this approach is that calling parent to invoke the parent class constructor would result in the creation of a lock of type Synchronized. This is exactly the kind of problem which dealing with cross-cutting concerns is meant to avoid. Instead, whenever a derived class is created, any associated environment class may also be replaced by another (not necessarily derived) class, as illustrated in this example:

```
class interface  CountSynchBuffer
extends SynchBuffer
  // constructor for object of this class
  initial (int count);
  in Semaphored Object read ();
  in Semaphored write (Object newData);
endc;

class implementation SynchBuffer
  initial (int count)
    maxProcs = count;
    parent ();
  endm;
  // read, write now controlled by a
  // semaphore -- otherwise same
  in Semaphored s initial ()
    s = new Semaphored (maxProcs);
  endm;
  in s Object read ()
    return parent ();
  endm;
  in s write (Object newData)
    parent (newData);
  endm;
  data
    int maxProcs;
endc;
```

In this example, possibly more than one concurrent reader and writer is permitted, so a semaphore is used instead of a lock. It should now be clear why initializing the

[1]This example illustrates the value of separating class definitions and implementations, but this is a point worth debating further outside the context of this paper.

environment or environments is split off from the class's constructor. We cannot rely on the initialization in the `SynchBuffer` class, because a semaphore needs to have an initial value for its counter. Note also that although the environment (or environments) defines an (implicit) instance variable, it is *not* part of the class in the sense that a derived class can replace it by something complete different.

3.3 Error Handling

Error handling introduces one new mechanism: terminating a method early. An error handler may either return a valid value (though terminating the current method and its environment) and allow execution to continue, or terminate the program (further options for propagation are possible but not explored). Early termination is achieved by calling `exit` in the error handler, in one of four forms:

- *kill* – first parameter `terminate`, indicating the program should terminate; if the final parameter is non-`null`, it contains information on the error
- *propagate* – first parameter `propagate`, indicating the method should terminate and call `catch` in the environment where it was called; the return value may be set as in the next two cases
- *return value* – first parameter `continue`, second parameter `true`, last parameter a reference to an object to return
- *return but no return value* – first parameter `continue`, second parameter `false`, last parameter ignored; return from the method without returning a value

In the first case, the program is terminated. Normally, a class will not define `exit` itself, since `exit` isn't called explicitly, and its behaviour includes a nonstandard operation (early termination). If a class does define `exit`, it should always call `parent` as its last step.

An error is signalled by calling `catch` with a parameter which is conventionally of a class derived from `Error` (though there is no reason in principle not to use another class, as long as the environment could handle it). If the environment has no `catch` method defined for the class of the parameter, the class is checked for a suitable `catch` method. If none is found, the program is terminated. An error handler can elect either to complete the method which invoked it by returning, having the effect of exiting from the method, or terminating the program. In the former case, the `exit` method of the environment (if defined, otherwise, of the class) is called.

As currently conceived, at most one error and one ordinary environment may be used. Nested environments are an idea worth considering, e.g., to allow multiple error handlers to be installed, but introduce more complexity than is useful for a an initial exploration of an idea.

3.4 Summary

The examples presented here have not explored all possible variations. However, they do give a general idea of the possibilities. Adding another dimension of abstraction across the inheritance hierarchy has been illustrated in a number of forms, with flexibility to implement environments in a number of ways. Cross-cutting concerns such as synchronization can be added orthogonally to the class hierarchy and can either be specialized in tandem with or independently of the class hierarchy.

4 Preliminary Evaluation

This section provides a preliminary evaluation of the idea of mutable environments, based on the examples previously explored. Ad hoc extension, class-based extension and error handling are relatively simple mechanisms in terms of the limited extra notation required. A few new keywords are needed: `mutable` and `in`. A notation is needed to separate out environment constructors in a class, and a mechanism for early termination of a method is required for error handling.

This level of simplicity does not necessarily imply simplicity in use, or simplicity of implementation.

Simplicity in use is difficult to evaluate without an implementation capable of developing a wide range of examples. The examples provided here are not complex, but do not in themselves illustrate generality or applicability, as they represent a limited range of cases.

Although multiple environments are not a feature of the initial design, some consideration is given to how they could be added.

Since evaluation of applicability and usability of the approach requires an implementation, the remainder of this section examines general implementation issues, then issues for class extension and ad hoc extension. In all cases, implementation issues are explored by considering the kind of simple extensions of an existing language which could be achieved with a preprocessor, as a minimal standard for evaluating implementability. Finally, issues identified in this section are summarized.

4.1 General implementation issues

In languages with a top-level class (commonly called `Object`), a new class, `EnvObject`, could be derived from `Object`. All classes not explicitly derived from a class would be modified to be derived from `EnvObject`, and any which were explicitly derived from `Object` modified to be derived from the new class. In languages without a top-level class, a new top-level class could be defined, and all classes not derived from another class be modified to be derived from this class. This new class would define defaults for standard mutable methods.

It would also be necessary to define a default environment class, from which others would be derived, and a default error handler class, as well as a default for class

`Error`.

All methods would be redefined to include an additional pair of parameters representing the optional environment and error handler (if multiple environments were permitted, a list of environments could be used). The methods would be patched to check if the environments existed (not `null`). Calls to the environment `start` and `exit` would have to be patched in, to be executed if the test for the paremeters succeeded. Calls without any explicit environment would have `null` added as the additional parameters.

4.2 Ad hoc extension

Ad hoc extension can be implemented by using the extra environment parameters directly. A call such as

```
in new Synchronized ()
  object.method  (parameter);
```

which has an environment but not an error handler environment would be implemented as

```
object.method  (parameter,
  new Synchronized ( ), null );
```

An error handler environment would replace the `null` final parameter. If multiple environments were implemented, adding each to a list would replace the addition of a fixed number of extra parameters.

4.3 Class extension

Patching classes requires changes in method implementations, as well as the changes already outlined for the interfaces. Each method call is one similar to that used in ad hoc extension, except the environment would be patched in from that defined in the class. For example, to implement the `read` method in the `SynchBuffer` implementation using an object called `lock` – illustrating changes always made and changes specific this example:

```
Object read (Environment env, ErrorHandler err)
  env = lock; // for class-defined environment
  if (env != null)  // always patch in
    env.start (); // always patch in
  Object result = parent ();  // always change
  // always patch in next 2 lines
  if (env != null)
    env.exit (continue, true, result);
  // exit call actually patched here
  // to avoid complications of how to
  // return a value 1 layer
  // of call away
  return result;  // always add
endm;
```

A practical issue in constructing a call is that the environment's instance variable is defined in the class and hence is not visible in the rest of the program. To work around this problem, each class would have to define an additional version of each method without the extra parameters, which called the rewritten version using the class-defined environment.

Note that this example says nothing about error handling. Error handling results in a specific call to a version of `catch` with a suitable parameter. While, logically, the final return is done in `exit`, the same effect can be achieved by placing a `return` after every call of `exit`.

Handling errors is somewhat more complex, and some additional detail needs to be worked through – but some aspects are dependent on the language on top of which these changes are layered. For example, in a language with exceptions, the error-handling mechanism could be patched in to exception handlers. Since the error handler environment is passed to all methods, they could follow the same logic as this example – check if the handler was non-`null`, and if so, use it. If not, call `catch` on the current class.

The most complicated issues relate to returning prematurely. Current languages do not in general have a mechanism allowing a called method to terminate its caller, except via exceptions. Forcing a return value to be returned from one level of call deeper than the method where the type of return is defined is also not a feature of common languages.

4.4 Evaluation Summary

Some of the simpler cases can be handled with relatively minor transformations to a standard object-oriented language. A preprocessor could be used to prototype these changes without designing a whole new language. Full implementation of the ideas requires some refinement of the design, to simplify details. However, the relatively mechanical changes identified so far make the possibility of implementation realistic.

Nothing considered in this section requires features unusual in a typical object-oriented language. Approaches to adding error handling would be more diverse: languages with support for exception handling, for example, would provide a different basis for implementing error handling to those without. Without exception handling, it would be necessary to patch in the kind of logic programmers dread creating manually to check for errors after every method returned. However, this patching should still be relatively mechanical.

Adding these features onto an existing language is a simple test-of-concept approach to evaluating implementability. Designing a whole new language is not essential to developing the idea further – an extension of existing languages would be less work. However, should the ideas prove viable after more detailed investigation, it may be worth investigating how to implement a language taking these ideas forward without unnecessary features of some other underlying language.

5 Conclusions

The idea of mutable environments has promise, but needs further exploration. The variations explored in this paper meet the major objectives of allowing cross-cutting concerns to be addressed orthogonally to the class hierarchy,

while allowing specialization of the resulting abstractions. These specializations can either follow the class hierarchy or their own, independent hierarchy.

The ideas developed here are comparable to aspect-oriented programming and Timor bracket methods, but provide an alternative approach. Whether this alternative approach will be simpler in practice requires further exploration. However, given the interest in aspect-oriented programming and the fact that the design space is not fully explored, it is worth examining alternative ideas with similar objectives.

While many similar ideas exist, the approach here has been to simplify mechanisms down as far as possible. This simplification has two goals: simplifying implementation, and simplifying usage. Both of these goals require further work to evaluate fully. While some aspects of implementation, as outlined here, appear straightforward (if simple alterations to a typical existing language is a guide), some details, including error handling, require further analysis.

Simplicity in use is difficult to evaluate without a complete language. Consequently, it will be important in follow-up work to fill gaps in the design outlined here, and to evaluate the concepts across a wide range of examples.

A test of the value of an idea is whether it leads in directions other than the original motivating examples. Future work on mutable environments will evaluate possible unexpected consequences. For example, an aspect of mutable environments which has not been explored is the fact that a class used as an environment need not be designed as such. This creates the possibility to include any class as a building block in another class design, with all the flexibility of the approach defined here. For example, a class defining dates could be used as an environment class, rather than embedding it in the class as an instance method. The value in this approach is that the abstraction which this date class defines can be limited by only defining a subset of its methods as mutable methods in the mutable class. A different class using the same methods, but having other functionality (or efficiencies), could replace the original date class. By limiting exposure to a class's public interface, mutable environments have the potential to make designs more flexible, and less dependent on inessential details of components.

Overall, mutable environments appear to have promise as an extension of the abstraction mechanisms offered by object-oriented programming.

References

[1] Ken Auer. Smalltalk training: As innovative as the environment. *Comm. ACM*, 38(10):115–117, October 1995.

[2] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin locks: featherweight synchronization for java. In *Proc. ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 258–268, Montreal, Quebec, Canada, 1998. ACM Press.

[3] William Berg, Marshall Cline, and Mike Girou. Lessons learned from the OS/400 OO project. *Comm. ACM*, 38(10):54–64, October 1995.

[4] Christophe Dony. Exception handling and object-oriented programming: towards a synthesis. In *Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 322–330, Ottawa, Canada, 1990. ACM Press.

[5] Mohamed E Fayad and Wei-Tek Tsai. Object-oriented experiences. *Comm. ACM*, 38(10):51–53, October 1995.

[6] Frakes and Fox. Sixteen questions about software reuse. *Comm. ACM*, 38(6):75–87,112, June 1995.

[7] Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *Proc. 2nd Int. Conf. on Aspect-oriented software development*, pages 60–69. ACM Press, 2003.

[8] J. L. Keedy, K. Espenlaub, G. Menger, and C. Heinlein. Qualifying types with bracket methods in Timor. *Journal of Object Technology*, January/February 2004. to appear.

[9] J. L. Keedy, G. Menger, C. Heinlein, and F. Henskens. Qualifying types illustrated by synchronisation examples. In M. Aksit, M. Mezini, and R. Unland, editors, *Objects, Components, Architectures, Services and Applications for a Networked World, International Conference NetObjectDays, NODe 2002*, volume LNCS 2591, pages 330–344, Erfurt, Germany, 2003. Springer.

[10] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proc. European Conference on Object-Oriented Programming*, pages 220–242, Jyväskylä, Finland, 1997.

[11] M. M. Lehman and J. F. Ramil. Evolution in software and related areas. In *Proc. 4th Int. workshop on Principles of software evolution*, pages 1–16, Vienna, Austria, 2002. ACM Press.

[12] Martin Lippert and Cristina Videira Lopes. A study on exception detecton and handling using aspect-oriented programming. In *Proceedings of the 22nd international conference on Software engineering*, pages 418–427, Limerick, Ireland, 2000. ACM Press.

[13] Philip Machanick. The abstraction-first approach to data abstraction and algorithms. *Computers & Education*, 31(2):135–150, September 1998.

[14] Linda M. Seiter, Jens Palsberg, and Karl J. Lieberherr. Evolution of object behavior using context relations. In *Proc. 4th ACM SIGSOFT Symp. on the Foundations of Software Engineering*, pages 46–57, 1996.

[15] K. Watsen and M. Zyda. Bamboo – a portable system for dynamically extensible, real-time, networked, virtual environments. In *Proc. IEEE Virtual Reality Annual International Symposium (VRAIS'98)*, pages 252–260, Atlanta, Georgia, March 1998.