Scheme with Classes, Mixins, and Traits

Matthew Flatt[1], Robert Bruce Findler[2], and Matthias Felleisen[3]

[1] University of Utah
[2] University of Chicago
[3] Northeastern University

Abstract. The Scheme language report advocates language design as the composition of a small set of orthogonal constructs, instead of a large accumulation of features. In this paper, we demonstrate how such a design scales with the addition of a class system to Scheme. Specifically, the PLT Scheme class system is a collection of orthogonal linguistic constructs for creating classes in arbitrary lexical scopes and for manipulating them as first-class values. Due to the smooth integration of classes and the core language, programmers can express mixins and traits, two major recent innovations in the object-oriented world. The class system is implemented as a macro in terms of procedures and a record-type generator; the mixin and trait patterns, in turn, are naturally codified as macros over the class system.

1 Growing a Language

The Revised[5] Report on the Scheme programming language [20] starts with the famous proclamation that "[p]rogramming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary." As a result, Scheme's core expression language consists of just six constructs: variables, constants, conditionals, assignments, procedures, and function applications. Its remaining constructs implement variable definitions and a few different forms of procedure parameter specifications. Everything else is defined as a function or macro.

PLT Scheme [25], a Scheme implementation intended for language experimentation, takes this maxim to the limit. It extends the core of Scheme with a few constructs, such as modules and generative structure definitions, and provides a highly expressive macro system. Over the past ten years, we have used this basis to conduct many language design experiments, including the development of an expressive and practical class system. We have designed and implemented four variants of the class system, and we have re-implemented DrScheme [13]—a substantial application of close to 200,000 lines of PLT Scheme code—in terms of this class system as many times.

Classes in PLT Scheme are first-class values, and the class system's scoping rules are consistent with Scheme's lexical scope and single namespace. Furthermore, the class system serves as a foundation for further macro-based explorations into class-like mechanisms, such as mixins and traits.

A mixin [11] is a class declaration parameterized over its superclass using `lambda`. Years of experience with these mixins shows that they are practical. Scoping rules for

methods allow both flexibility and control in combining mixins, while explicit inheritance specifications ensure that unintentional collisions are flagged early.

In this setting, a trait [29] is a set of mixins. Although mixins and traits both represent extensions to a class, we distinguish traits from mixins, because traits provide fine-grained control over individual methods in the extension, unlike mixins.

Last but not least, objects instantiated by the class system are efficient in space and time, whether the class is written directly or instantiated through mixins and or traits. In particular, objects in our system consume a similar amount of space to a Smalltalk or Java object. Method calls have a cost similar to Smalltalk method calls or interface-based Java calls. In short, the class system is efficient as well as effective.

2 Classes

In PLT Scheme, a `class` expression denotes a first-class value, just like a `lambda` expression:

`(class` *superclass-expr decl-or-expr**`)`

The *superclass-expr* determines the superclass for the new class. Each *decl-or-expr* is either a declaration related to methods, fields, and intialization arguments, or it is an expression that is evaluated each time that the class is instantiated. In other words, instead of a method-like constructor, a class has initialization expressions interleaved with field and method declarations. Figure 1 displays a simplified grammar for *decl-or-expr*.

By convention, class names end with `%`. The built-in root class is `object%`. Thus the following expression creates a class with public methods *`get-size`*, *`grow`*, and `eat`:

```
(class object%
  (init size)                ; initialization argument
  (define current-size size) ; field
  (super-new)                ; superclass initialization
  (define/public (get-size)
    current-size)
  (define/public (grow amt)
    (set! current-size (+ amt current-size)))
  (define/public (eat other-fish)
    (grow (send other-fish get-size))))
```

The *`size`* initialization argument must be supplied via a named argument when instantiating the class through the `new` form:

```
(new (class object% (init size) ...) [size 10])
```

Of course, we can also name the class and its instance:

```
(define fish% (class object% (init size) ...))
(define charlie (new fish% [size 10]))
```

In the definition of *`fish%`*, *`current-size`* is a private field that starts out with the value of the *`size`* initialization argument. Initialization arguments like *`size`* are available

decl-or-expr	::= `(define` *id expr*`)`	private field definition
	| `(define/`*method-spec* `(`*method-id id**`)` *expr**`)`	method definition
	| `(init` *id-with-expr**`)`	initialization argument
	| `(field` *id-with-expr**`)`	public field
	| `(inherit` *method-id**`)`	inherit method, for direct access
	| *expr*	initialization expression
method-spec	::= `public`	new method
	| `override`	override method
	| `private`	private method
id-with-expr	::= *id*	without initial value or default
	| `[`*id expr*`]`	with initial value or default
expr	::= `(new` *class-expr* `[`*id expr*`]`*`)`	object creation
	| `(send` *object-expr method-id expr**`)`	external method call
	| `(`*method-id expr**`)`	internal method call (in `class`)
	| `this`	object self-reference (in `class`)
	| `(super` *method-id expr**`)`	call overridden method (in `class`)
	| `(super-new [`*id expr*`]`*`)`	call super initialization (in `class`)
	| ...	all other Scheme expression forms

superclass-expr, *class-expr*, and *object-expr* are aliases for *expr*; *method-id* is an alias for *id*

Fig. 1. Simplified PLT Scheme class system grammar

only during class instantiation, so they cannot be referenced directly from a method. The *current-size* field, in contrast, is available to methods.

The `(super-new)` expression in *fish%* invokes the initialization of the superclass. In this case, the superclass is `object%`, which takes no initialization arguments and performs no work; `super-new` must be used, anyway, because a class must always invoke its superclass's initialization.

Initialization arguments, field declarations, and expressions such as `(super-new)` can appear in any order within a `class`, and they can be interleaved with method declarations. The relative order of expressions in the class determines the order of evaluation during instantiation. For example, if a field's initial value requires calling a method that works only after superclass initialization, then the field declaration is placed after the `super-new` call. Ordering field and initialization declarations in this way helps avoid imperative assignment. The relative order of method declarations makes no difference for evaluation, because methods are fully defined before a class is instantiated.

2.1 Methods

Each of the three `define/public` declarations in *fish%* introduces a new method. The declaration uses the same syntax as a Scheme function, but a method is not accessible as an independent function. A call to the *grow* method of a *fish%* object requires the `send` form:

```
(send charlie grow 6)
(send charlie get-size) ; ⇒ 16
```

Within *fish%*, self methods can be called like functions, because the method names are in scope. For example, the *eat* method within *fish%* directly invokes the *grow* method. Within a class, attempting to use a method name in any way other than a method call results in a syntax error.

In some cases, a class must call methods that are supplied by the superclass but not overridden. In that case, the class can use `send` with `this` to access the method:

```
(define hungry-fish% (class fish% (super-new)
                       (define/public (eat-more fish1 fish2)
                         (send this eat fish1)
                         (send this eat fish2))))
```

Alternately, the class can declare the existence of a method using `inherit`, which brings the method name into scope for a direct call:

```
(define hungry-fish% (class fish% (super-new)
                       (inherit eat)
                       (define/public (eat-more fish1 fish2)
                         (eat fish1) (eat fish2))))
```

With the `inherit` declaration, if *fish%* had not provided an *eat* method, an error would be signaled in the evaluation of the `class` form for *hungry-fish%*. In contrast, with `(send this ...)`, an error would not be signaled until the *eat-more* method is called and the `send` form is evaluated. For this reason, `inherit` is preferred.

Another drawback of `send` is that it is less efficient than `inherit`. Invocation of a method via `send` involves finding a method in the target object's class at run time, making `send` comparable to an interface-based method call in Java. In contrast, `inherit`-based method invocations use an offset within the class's method table that is computed when the class is created.

To achieve performance similar to `inherit`-based method calls when invoking a method from outside the method's class, the programmer must use the `generic` form, which produces a class- and method-specific *generic method* to be invoked with `send-generic`:

```
(define get-fish-size (generic fish% get-size))
(send-generic charlie get-fish-size) ; ⇒ 16
(send-generic (new hungry-fish% [size 32]) get-fish-size) ; ⇒ 32
(send-generic (new object%) get-fish-size) ; Error: not a fish%
```

Roughly speaking, the form translates the class and the external method name to a location in the class's method table. As illustrated by the last example, sending through a generic method checks that its argument is an instance of the generic's class.

Whether a method is called directly within a `class`, through a generic method, or through `send`, method overriding works in the usual way:

```
(define picky-fish% (class fish% (super-new)
                      (define/override (grow amt)
                        ;; Doesn't eat all of its food
                        (super grow (* 3/4 amt)))))
(define daisy (new picky-fish% [size 20]))
(send daisy eat charlie) ; charlie's size is 16
(send daisy get-size) ; ⇒ 32
```

The *grow* method in *picky-fish%* is declared with `define/override` instead of `define/public`, because *grow* is meant as an overriding declaration. If *grow* had been declared with `define/public`, an error would have been signaled when evaluating the `class` expression, because *fish%* already supplies *grow*.

Using `define/override` also allows the invocation of the overridden method via a `super` call. For example, the *grow* implementation in *picky-fish%* uses `super` to delegate to the superclass implementation.

2.2 Initialization Arguments

Since *picky-fish%* declares no initialization arguments, any initialization values supplied in `(new picky-fish% ...)` are propagated to the superclass initialization, i.e., to *fish%*. A subclass can supply additional initialization arguments for its superclass in a `super-new` call, and such initialization arguments take precedence over arguments supplied to `new`. For example, the following *size-10-fish%* class always generates fish of size 10:

```
(define size-10-fish% (class fish% (super-new [size 10])))
(send (new size-10-fish%) get-size) ; ⇒ 10
```

In the case of *size-10-fish%*, supplying a *size* initialization argument with `new` would result in an initialization error; because the *size* in `super-new` takes precedence, a *size* supplied to `new` would have no target declaration.

An initialization argument is optional if the `class` form declares a default value. For example, the following *default-10-fish%* class accepts a *size* initialization argument, but its value defaults to 10 if no value is supplied on instantiation:

```
(define default-10-fish% (class fish%
                           (init [size 10])
                           (super-new [size size])))
(new default-10-fish%) ; ⇒ a fish of size 10
(new default-10-fish% [size 20]) ; ⇒ a fish of size 20
```

In this example, the `super-new` call propagates its own *size* value as the *size* initialization argument to the superclass.

2.3 Internal and External Names

The two uses of *size* in *default-10-fish%* expose the double life of class-member identifiers. When *size* is the first identifier of a bracketed pair in `new` or `super-new`, *size* is an *external name* that is symbolically matched to an initialization argument in a class. When *size* appears as an expression within *default-10-fish%*, *size* is an *internal name* that is lexically scoped. Similarly, a call to an inherited *eat* method uses *eat* as an internal name, whereas a `send` of *eat* uses *eat* as an external name.

The full syntax of the `class` form allows a programmer to specify distinct internal and external names for a class member. Since internal names are local, they can be α-renamed to avoid shadowing or conflicts. Such renaming is not frequently necessary, but workarounds in the absence of α-renaming can be especially cumbersome.

2.4 Interfaces

Interfaces are useful for checking that an object or a class implements a set of methods with a particular (implied) behavior. This use of interfaces is helpful even without a static type system (which is the main reason that Java has interfaces).

An interface in PLT Scheme is created using the `interface` form, which merely declares the method names required to implement the interface. An interface can extend other interfaces, which means that implementations of the interface automatically implement the extended interfaces.

`(interface (`*superinterface-expr**`) `*id**`)`

To declare that a class implements an interface, the `class*` form must be used instead of `class`:

`(class*` *superclass-expr* `(`*interface-expr**`)` *decl-or-expr**`)`

For example, instead of forcing all fish classes to be derived from *`fish%`*, we can define *`fish-interface`* and change the *`fish%`* class to declare that it implements *`fish-interface`*:

```
(define fish-interface (interface () get-size grow eat))
(define fish% (class* object% (fish-interface) ...))
```

If the definition of *`fish%`* does not include *`get-size`*, *`grow`*, and `eat` methods, then an error is signaled in the evaluation of the `class*` form, because implementing the *`fish-interface`* interface requires those methods.

The `is-a?` predicate accepts either a class or interface as its first argument and an object as its second argument. When given a class, `is-a?` checks whether the object is an instance of that class or a derived class. When given an interface, `is-a?` checks whether the object's class implements the interface. In addition, the `implementation?` predicate checks whether a given class implements a given interface.

2.5 Final, Augment, and Inner

As in Java, a method in a `class` form can be specified as *final*, which means that a subclass cannot override the method. A final method is declared using `public-final` or `override-final`, depending on whether the declaration is for a new method or an overriding implementation.

Between the extremes of allowing arbitrary overriding and disallowing overriding entirely, the class system also supports Beta-style *augmentable* methods [22]. A method declared with `pubment` is like `public`, but the method cannot be overridden in subclasses; it can be augmented only. A `pubment` method must explicitly invoke an augmentation (if any) using `inner`; a subclass augments the method using `augment`, instead of `override`.

In general, a method can switch between augment and override modes in a class derivation. The `augride` method specification indicates an augmentation to a method where the augmentation is itself overrideable in subclasses (though the superclass's implementation cannot be overridden). Similarly, `overment` overrides a method and makes the overriding implementation augmentable. Our earlier work [19] motivates and explains these extensions and their interleaving.

2.6 Controlling the Scope of External Names

As noted in Section 2.3, class members have both internal and external names. A member definition binds an internal name locally, and this binding can be locally α-renamed. External names, in contrast, have global scope by default, and a member definition does not bind an external name. Instead, a member definition refers to an existing binding for an external name, where the member name is bound to a *member key*; a class ultimately maps member keys to methods, fields, and initialization arguments.

Recall the *hungry-fish%* `class` expression:

```
(define hungry-fish% (class fish% ...
                       (inherit eat)
                       (define/public (eat-more fish1 fish2)
                         (eat fish1) (eat fish2))))
```

During its evaluation, the *hungry-fish%* and *fish%* classes refer to the same global binding of *eat*. At run time, calls to *eat* in *hungry-fish%* are matched with the *eat* method in *fish%* through the shared method key that is bound to *eat*.

The default binding for an external name is global, but a programmer can introduce an external-name binding with the `define-member-name` form.

(`define-member-name` *id member-key-expr*)

In particular, by using (`generate-member-key`) as the *member-key-expr*, an external name can be localized for a particular scope, because the generated member key is inaccessible outside the scope. In other words, `define-member-name` gives an external name a kind of package-private scope, but generalized from packages to arbitrary binding scopes in Scheme.

For example, the following *fish%* and *pond%* classes cooperate via a *get-depth* method that is only accessible to the cooperating classes:

```
(define-values (fish% pond%) ;; two mutually recursive classes
  (let () ; create a local definition scope
    (define-member-name get-depth (generate-member-key))
    (define fish%
      (class ... (define my-depth ...)
                 (define my-pond ...)
                 (define/public (dive amt)
                   (set! my-depth
                     (min (+ my-depth amt)
                          (send my-pond get-depth))))))
    (define pond%
      (class ... (define current-depth ...)
                 (define/public (get-depth) current-depth)))
    (values fish% pond%)))
```

External names are in a namespace that separates them from other Scheme names. This separate namespace is implicitly used for the method name in `send`, for initialization-argument names in `new`, or for the external name in a member definition. The special `member-name-key` provides access to the binding of an external name in an arbitrary expression position: (`member-name-key` *id*) form produces the member-key binding of *id* in the current scope.

A member-key value is primarily used on with a `define-member-name` form. Normally, then, (`member-name-key` *id*) captures the method key of *id* so that it can be communicated to a use of `define-member-name` in a different scope. This capability turns out to be useful for generalizing mixins (see Section 3.4).

2.7 Implementation of Classes

The `class` form is implemented in terms of a primitive `make-struct-type` procedure, which generates a data type that is distinct from all existing data types. The new data type's specification includes the number of slots that should be allocated for instances of the data type, plus properties for the data type. A class corresponds to a fresh data type with one slot for each field and with a property for the class's method table.

Most of the compile-time work for the `class` macro is in expanding the individual expressions and declarations in the method body, and ensuring that the declarations are locally consistent (e.g., no duplicate method declarations). Indeed, of the roughly 3,500 lines of Scheme code that implement the class system, $3/4$ implement compile-time work (especially syntax checking to provide good error messages), and $1/4$ of the lines implement run-time support.

The run-time representation of a class includes the method implementations—as procedures transformed to take an explicit `this` argument—and information about introduced methods and expected superclass methods. The run-time work of class creation mostly checks the consistency of the class extensions with a supplied superclass, closes the method implementations with specific methods for `super` calls, and closes method implementations with specific `vtable` indices for direct method calls.

3 Mixins

Since `class` is an expression form instead of a top-level declaration as in Smalltalk and Java, a `class` form can be nested inside any lexical scope, including `lambda`. The result is a *mixin*, i.e., a class extension that is parameterized with respect to its superclass [11].

For example, we can parameterize the *picky-fish%* class over its superclass to define *picky-mixin*:

```
(define (picky-mixin %)
  (class % (super-new)
    (define/override (grow amt) (super grow (* 3/4 amt)))))
(define picky-fish% (picky-mixin fish%))
```

Many small differences between Smalltalk-style classes and our classes contribute to the effective use of mixins. In particular, the use of `define/override` makes explicit that *picky-mixin* expects a class with a *grow* method. If *picky-mixin* is applied to a class without a *grow* method, an error is signaled as soon as *picky-mixin* is applied.

Similarly, a use of `inherit` enforces a "method existence" requirement when the mixin is applied:

```
(define (hungry-mixin %)
  (class % (super-new)
```

```
  (inherit eat)
  (define/public (eat-more fish1 fish2) (eat fish1) (eat fish2))))
```

The advantage of mixins is that we can easily combine them to create new classes whose implementation sharing does not fit into a single-inheritance hierarchy—without the ambiguities associated with multiple inheritance. Equipped with `picky-mixin` and `hungry-mixin`, creating a class for a hungry, yet picky fish is straightforward:

```
(define picky-hungry-fish% (hungry-mixin (picky-mixin fish%)))
```

The use of keyword initialization arguments is critical for the easy use of mixins. For example, `picky-mixin` and `hungry-mixin` can augment any class with suitable `eat` and `grow` methods, because they do not specify initialization arguments and add none in their `super-new` expressions:

```
(define person% (class object%
                  (init name age)
                  ...
                  (define/public (eat food) ...)
                  (define/public (grow amt) ...)))
(define child% (hungry-mixin (picky-mixin person%)))
(define oliver (new child% [name "Oliver"][age 6]))
```

Finally, the use of external names for class members (instead of lexically scoped identifiers) makes mixin use convenient. Applying `picky-mixin` to `person%` works because the names `eat` and `grow` match, without any a priori declaration that `eat` and `grow` should be the same method in `fish%` and `person%`. This feature is a potential drawback when member names collide accidentally; some accidental collisions can be corrected by limiting the scope external names, as discussed in Section 2.6.

3.1 Mixins and Interfaces

Using `implementation?`, `picky-mixin` could require that its base class implements `grower-interface`, which could be implemented by both `fish%` and `person%`:

```
(define grower-interface (interface () grow))
(define (picky-mixin %)
  (unless (implementation? % grower-interface)
    (error "picky-mixin: not a grower-interface class"))
  (class % ...))
```

Another use of interfaces with a mixin is to tag classes generated by the mixin, so that instances of the mixin can be recognized. In other words, `is-a?` cannot work on a mixin represented as a function, but it can recognize an interface (somewhat like a *specialization interface* [21]) that is consistently implemented by the mixin. For example, classes generated by `picky-mixin` could be tagged with `picky-interface`, enabling the `is-picky?` predicate:

```
(define picky-interface (interface ()))
(define (picky-mixin %)
  (unless (implementation? % grower-interface)
    (error "picky-mixin: not a grower-interface class"))
```

```
  (class* % (picky-interface) ...))
(define (is-picky? o)
  (is-a? o picky-interface))
```

3.2 The Mixin Macro

To codify the `lambda`-plus-`class` pattern for implementing mixins, including the use of interfaces for the domain and range of the mixin, PLT Scheme's class system provides a `mixin` macro:

`(mixin (`*interface-expr**`) (`*interface-expr**`)` *decl-or-expr**`)`

The first set of *interface-expr*s determines the domain of the mixin, and the second set determines the range. That is, the expansion is a function that tests whether a given base class implements the first sequence of *interface-expr*s and produces a class that implements the second sequence of *interface-expr*s. Other requirements, such as the presence of `inherit`ed methods in the superclass, are then checked for the `class` expansion of the `mixin` form.

3.3 Mixins, Augment, and Inner

Mixins not only override methods and introduce public methods, they can also augment methods, introduce augment-only methods, add an overrideable augmentation, and add an augmentable override — all of the things that a class can do (see Section 2.5).

Bracha and Cook [11] observed that mixins alone can express both Smalltalk-style method overriding and Beta-style method augmenting, depending on the order of mixin composition. Their result, however, depends on choosing an order of composition; otherwise, the security benefits of Beta-style augmenting are lost (as we have observed [19] to be the case for gbeta). Our goal in adding `augment` and `inner` to the class system is to provide the same sort of security guarantees as Beta, which explains why we implement mixins in terms of classes, not classes in terms of mixins.

3.4 Parameterized Mixins

As noted in Section 2.6, external names can be bound with `define-member-name`. This facility allows a mixin to be generalized with respect to the methods that it defines and uses. For example, we can parameterize *`hungry-mixin`* with respect to the external member key for *`eat`*:

```
(define (make-hungry-mixin eat-method-key)
  (define-member-name eat eat-method-key)
  (mixin () () (super-new)
    (inherit eat)
    (define/public (eat-more x y) (eat x) (eat y))))
```

To obtain a particular hungry-mixin, we must apply this function to a member key that refers to a suitable *`eat`* method, which we can obtain using `member-name-key`:

```
((make-hungry-mixin (member-name-key eat))
 (class object% ... (define/public (eat x) 'yum)))
```

Above, we apply *hungry-mixin* to an anonymous class that provides *eat*, but we can also combine it with a class that provides *chomp*, instead:

```
((make-hungry-mixin (member-name-key chomp))
 (class object% ... (define/public (chomp x) 'yum)))
```

4 Traits

A *trait* [28, 29] is similar to a mixin, in that it encapsulates a set of methods to be added to a class. A trait is different from a mixin in that its individual methods can be manipulated with trait operators such as `sum` (merge the methods of two traits), `exclude` (remove a method from a trait), and `alias` (add a copy of a method with a new name; do not redirect any calls to the old name). The practical difference between mixins and traits is that two traits can be combined, even if they include a common method and even if neither method can sensibly override the other. In that case, the programmer must explicitly resolve the collision, usually by aliasing methods, excluding methods, and merging a new trait that uses the aliases.

Suppose our *fish%* programmer wants to define two class extensions, *spots* and *stripes*, each of which includes a *get-color* method. The fish's spot color should not override the stripe color nor vice-versa; instead, a *spots+stripes-fish%* should combine the two colors, which is not possible if *spots* and *stripes* are implemented as plain mixins. If, however, *spots* and *stripes* are implemented as traits, they can be combined. First, we alias *get-color* in each trait to a non-conflicting name. Second, the *get-color* methods are removed from both and the traits with only aliases are merged. Finally, the new trait is used to create a class that introduces its own *get-color* method based on the two aliases, producing the desired *spots+stripes* extension.

4.1 Traits as Sets of Mixins

One natural approach to implementing traits in PLT Scheme is as a set of mixins, with one mixin per trait method. For example, we might attempt to define the spots and stripes traits as follows, using association lists to represent sets:

```
(define spots-trait
  (list (cons 'get-color
              (lambda (%) (class % (super-new)
                                 (define/public (get-color) 'black))))))
(define stripes-trait
  (list (cons 'get-color
              (lambda (%) (class % (super-new)
                                 (define/public (get-color) 'red))))))
```

A set representation, such as the above, allows `sum` and `exclude` as simple manipulations; unfortunately, it does not support the `alias` operator. Although a mixin can be duplicated in the association list, the mixin has a fixed method name, e.g., *get-color*, and mixins do not support a method-rename operation. To support `alias`, we must parameterize the mixins over the external method name in the same way that *eat* was parameterized in Section 3.4.

4.2 Traits as Parameterized Mixins

To support the `alias` operation, *spots-trait* should be represented as:

```
(define spots-trait
  (list (cons (member-name-key get-color)
              (lambda (get-color-key %)
                (define-member-name get-color get-color-key)
                (class % (super-new)
                  (define/public (get-color) 'black))))))
```

When the *get-color* method in *spots-trait* is aliased to *get-trait-color* and the *get-color* method is removed, the resulting trait is the same as

```
(list (cons (member-name-key get-trait-color)
            (lambda (get-color-key %)
              (define-member-name get-color get-color-key)
              (class % (super-new)
                (define/public (get-color) 'black))))))
```

To apply a trait *T* to a class *C* and obtain a derived class, we use `(apply-trait T C)`. The `apply-trait` function supplies each mixin of *T* the key for the mixin's method and a partial extension of *C*:

```
(define (apply-trait T C)
  (foldr (lambda (m %) ((cdr m) (car m) %)) C T))
```

Thus, when the trait above is combined with other traits and then applied to a class, the use of *get-color* becomes a reference to the external name *get-trait-color*.

4.3 Inherit and Super in Traits

This first implementation of traits supports `alias`, and it supports a trait method that calls itself, but it does not support trait methods that call each other. In particular, suppose that a spot-fish's market value depends on the color of its spots:

```
(define spots-trait
  (list (cons (member-name-key get-color) ...)
        (cons (member-name-key get-price)
              (lambda (get-price %) ...
                (class % ...
                  (define/public (get-price) ... (get-color) ...))))))
```

In this case, the definition of *spots-trait* fails, because *get-color* is not in scope for the *get-price* mixin. Indeed, depending on the order of mixin application when the trait is applied to a class, the *get-color* method may not be available when *get-price* mixin is applied to the class. Therefore adding an `(inherit get-color)` declaration to the *get-price* mixin does not solve the problem.

One solution is to require the use of `(send this get-color)` in methods such as *get-price*. This change works because `send` always delays the method lookup until the method call is evaluated. The delayed lookup is more expensive than a direct call, however. Worse, it also delays checking whether a *get-color* method even exists.

A second, effective, and efficient solution is to change the encoding of traits. Specifically, we represent each method as a pair of mixins: one that introduces the method and one that implements it. When a trait is applied to a class, all of the method-introducing mixins are applied first. Then the method-implementing mixins can use `inherit` to directly access any introduced method.

```
(define spots-trait
  (list (list (local-member-name-key get-color)
              (lambda (get-color get-price %) ...
                (class % ...
                  (define/public (get-color) (void))))
              (lambda (get-color get-price %) ...
                 (class % ...
                   (define/override (get-color) 'black))))
        (list (local-member-name-key get-price)
              (lambda (get-price get-color %) ...
                (class % ...
                  (define/public (get-price) (void))))
              (lambda (get-color get-price %) ...
                (class % ...
                  (inherit get-color)
                  (define/override (get-price)
                    ... (get-color) ...))))))
```

With this trait encoding, `alias` works as in the Squeak implementation of traits. It adds a new method with a new name, but it does not change any references to the old method.

In contrast to the Squeak implementation [28], we can easily support a `rename` operation for traits with a bit of additional external-name parameterizations. Indeed, our `rename` operation even works for references in `inherit` and `send`.

Properly supporting `super` calls within a trait requires relatively little work when each `super` call to a method appears in an overriding implementation for the same method. In that case, no method-introducing mixin is needed, since overriding implies that the method exists already in the superclass. Special care is required if a `super` call is allowed in a method other than an overriding implementation, and a cycle of mutually `super`-calling methods may require an indirection to prevent a `super` call from accessing an implementation in the trait instead of the base class. Fortunately, the trait-application operator can generate this indirection automatically.

4.4 The Trait Macro

The general-purpose trait pattern is clearly too complex for a programmer to use directly, but it is easily codified in a `trait` macro:

`(trait (inherit` *id**`)`$^?$ `(define/`*method-spec* `(`*id id**`)` *expr**`)`*`)`

The *id*s in the optional `inherit` clause are available for direct reference in the method *expr*s, and they must be supplied either by other traits or the base class to which the trait is ultimately applied.

Using this form in conjunction with trait operators such as `sum`, `exclude`, `alias`, and `apply-trait`, we can implement *`spots-trait`* and *`stripes-trait`* as desired; see Figure 2.

```
(define spots-trait
  (trait
    (define/public (get-color) 'black)
    (define/public (get-price) ... (get-color) ...)))

(define stripes-trait
  (trait
    (define/public (get-color) 'red)))

(define spots+stripes-trait
  (sum (exclude (alias spots-trait get-color get-spots-color)
                get-color)
       (exclude (alias stripes-trait get-color get-stripes-color)
                get-color)
       (trait
         (inherit get-spots-color get-stripes-color)
         (define/public (get-color)
           ... (get-spots-color) ... (get-stripes-color) ...))))
```

Fig. 2. An example use of full-fledged traits

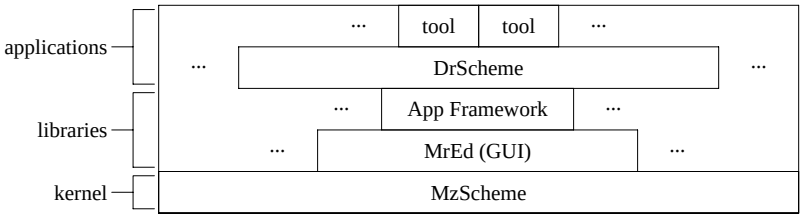

Fig. 3. PLT Scheme architecture

5 History and Experience

DrScheme is the most recognizable application that is built with PLT Scheme, and its implementation makes extensive use of the class system. Figure 3 shows how DrScheme fits into the architecture of PLT Scheme. *MzScheme* is the core compiler and run-time system, analogous to the JVM for Java. *MrEd* is the core GUI layer, analogous to AWT for Java. The *application framework* provides skeleton classes for typical kinds of GUI applications. Finally, DrScheme supports plug-in *tools* that extend the programming environment. (Ellipses in the figure represent other PLT libraries and applications.)

The language, kernel, and programming environment are sometimes difficult to distinguish, in part because they reinforce each other: MzScheme and MrEd were created as a platform to build DrScheme, and many programmers now choose PLT Scheme specifically because it is supported by DrScheme. Nevertheless, the distinctions are useful for understanding the uses of classes in DrScheme's implementation.

5.1 Current Uses of Classes

DrScheme employs classes primarily for its graphical interface, since the benefits of class-oriented programming are well understood for GUIs. In particular, the MrEd layer exports a class- and interface-based API for GUI programming, and it uses mixins internally to build most of the widget classes. The application framework layer exports a class-, interface-, and mixin-based API; the framework even includes classes with overrideable methods that act as mixins.

DrScheme's *editor* classes demonstrate many typical uses of classes and mixins. An editor represents the content of a window with interactive text and images:

Editors in MrEd Every editor implements the *editor<%>* interface, which has two base implementations: the *text%* class for a text-oriented, line-based layout, and the *pasteboard%* class for a free-form, two-dimensional layout.

The *text%* and *pasteboard%* classes are derived from more primitive, private variants *wx-text%* and *wx-pasteboard%*. The *wx-* variants share a superclass that implements common behavior at the primitive level, but *text%* and *pasteboard%* also share behavior that cannot be implemented in the primitive layer. Instead of duplicating refinements of *wx-text%* and *wx-pasteboard%*, the common refinements are implemented once in an internal mixin, thus creating a single point of control for shared behavior in *text%* and *pasteboard%*.

The *text%* and *pasteboard%* classes cooperate with the *editor-canvas%* class, which is instantiated to display an editor. Locally scoped external names serve the same role as package-private declarations to hide methods that are required for this inter-class cooperation.

Although most methods of *text%* and *pasteboard%* are overrideable, a few are augmentable only. For example, the *can-insert?* method is called before any insertion attempt to determine whether the editor can be modified. This method is augmentable only, which prevents a subclass from allowing insertions if a superclass (possibly defined by a more primitive layer) must disallow insertions to preserve invariants.

Editors in the Framework The application framework provides several editor mixins, such as an autosave mixin, a mixin to display editor state (such as the current line and column) into an information panel, and a mixin for chaining keymaps together. The framework also supplies nearly a dozen mixins that are specific to *text%*. The framework's top-level window class includes *get-editor%* and *get-canvas%* methods, so that a mixin for top-level windows can consistently extend the editor and canvas classes that are created for the window.

Certain editor and text mixins cooperate with a corresponding mixin for the display canvas. So far, we have mostly relied on naming conventions and run-time checks to help keep mixin applications in sync; we are considering implementing mixin layers [30] (via macros) for this purpose.

Editors in DrScheme A tool that extends to the DrScheme programming environment is implemented as a unit [16]. DrScheme supplies each tool unit with functions to register mixin refinements of its editors. That is, tool implementors get the same convenient API as the DrScheme implementors for extending the environment, even though tools can be mixed and matched in a given installation.

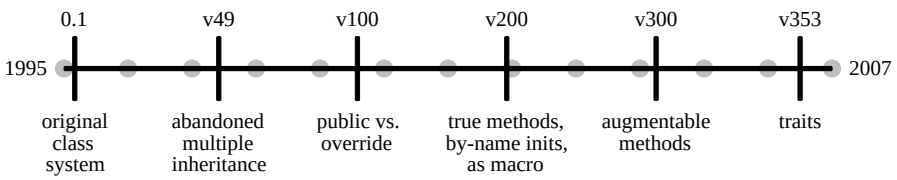

Fig. 4. PLT Scheme class system timeline

5.2 Language Evolution

Figure 4 shows how the class system in PLT Scheme has evolved over the project's 11-year history. To create the initial GUI base for DrScheme, we combined an embeddable Scheme system, libscheme [8], with a C++-based multi-platform GUI library, wxWindows [31]. We also added our own C++-based editor classes, which is why the GUI layer is called "MrEd." To make the C++ classes available in Scheme (for both class extension and instantiation), we extended libscheme with a built-in object system. As our changes to libscheme accumulated, we renamed it "MzScheme."

Our earliest design for classes included support for both mixins (as `class` plus `lambda`) and multiple inheritance of classes. We soon abandoned multiple inheritance, since it was rarely used, whereas mixins took hold early in our libraries.

For the first major re-design, we introduced the distinction between `public` methods and `override` methods. This avoided occasional confusion where a mixin application that was intended to introduce a method would instead override an existing method.

Through the first two major design stages, the class system implemented objects as records of closures, where a method is represented as a closure with `this` as a free variable. Such records are a typical way to represent objects in Scheme, and it worked well enough when objects were used in small quantities, such as objects for windows, buttons, and drawing pens. Over time, the addition of new kinds of snips to the editors, especially the nesting of text objects inside of editors, caused an overwhelming consumption of space and time.

The third major design abandoned methods as closures over `this` in favor of a more Smalltalk-like implementation where an object is a record of field values, plus a class-specific table of method procedures that accept an implicit `this` argument. This change eliminated performance problems related to the size of text objects in editors.

The third design also introduced by-name initialization arguments as an alternative to by-position arguments. As noted in Section 3, named initialization arguments complement mixin composition; in contrast, by-position arguments often force mixins to provide imperative initialization methods, since there is no simple way to distinguish optional initialization arguments for the mixin from initialization arguments intended for the superclass. In the current design, both forms of initialization arguments remain, but by-position arguments are used only in older libraries.

The first two implementations of classes were built into the language kernel. The implementation of the third design was greatly facilitated by MzScheme's switch from traditional Lisp macros to a modular macro system based on *`syntax-case`* [12, 15],

so that the class system could be implemented through macros instead of built into the kernel.

The relative ease of changing the macro-based implementation enabled the most recent major change to the class system, which was the addition of `augment` and `inner`. The change was motivated by bugs due to incorrect overriding of methods like *`can-insert?`*, especially within tools that extend DrScheme.

5.3 Open Issues

The PLT Scheme class system has evolved in response to ever more stringent requirements for stability, performance, and expressiveness. The regularity of events in Figure 4 is surprising—the tick marks correspond to actual dates when changes became widely deployed to users—but they match the consistent growth of PLT Scheme. Predicting further change (and, apparently, its timing) is easier than predicting the specific nature of the change, but several open issues are likely to attract attention.

The `class` forms's distinction between initialization arguments and fields makes explicit that values used only for initialization need not be stored in the object. Nevertheless, initialization arguments often turn into fields, and there seems to be no advantage in forcing programmers to explicitly designate such conversions; merely referencing an initialization argument from a method should be enough to convert it to a field. Automatic conversion, however, requires expanding all subexpressions when expanding a `class` form, but the `class` form needs to expand sub-expressions differently for fields than for initialization arguments. In other words, our macro technology affects our language design (in much the same way that parsing and type-checking concerns sometimes influence the outcome of other language design decisions).

In a similar vein, the class system prohibits an internal reference to a method that is not in an application position (i.e., as a method call). Occasionally, we would like to pass a method as a first-class value to functionals such as *`map`*. In this case, the `class` macro could easily convert the method to a closure over `this`; we instead force programmers to wrap the method with a `lambda` so that the closure allocation is more apparent. We may reconsider this design decision.

The run-time cost of object instantiation is higher than it should be. For an object with two initialization arguments that are both converted to fields, the instantiation time is a factor of 20 slower than for a comparable PLT Scheme record. The difference is in gathering and finding initializations arguments by name (which accounts for a factor of 10) and copying saved initialization arguments into fields (remaining factor of 2). One possible solution is to provide a form for specializing `new` in much the same way that `send-generic` specializes `send`.

Like most class systems, the PLT Scheme system conflates implementation inheritance and interface inheritance. That is, a subclass automatically implements any interface that its superclass implements. We are in a good position to try detaching interface inheritance from subclassing, but we have not yet explored that possibility.

Finally, although we have designed a class system that supports mixins and traits as separate extensions, the class system itself includes many built-in features that seem orthogonal: initialization protocols, several method overriding and augmenting protocols,

and both implementation and interface inheritance. Future work may uncover ways to remove weaknesses and restrictions, making our little pile of features even smaller.

6 Related Work on Classes in Scheme

Our approach of adding objects to Scheme closely resembles Friedman's [18] object-oriented style, but it also differs significantly from his work. The key difference concerns the instantiation of classes, which we separate from the macro expansion phase. Instead of specifying a class's method statically, we rely on a run-time computation to completely determine a class's shape. As a result, combining our `class` with `lambda` defines mixins that work on varieties of superclass shapes.

Historically, implementors of class systems for Scheme have used the message-passing metaphor literally, representing an object as a procedure that accepts a method-selecting symbol [1, 2]. More generally, Scheme programmers are often tempted to think of an object as a collection of closures, where `this` is built into each method's closure instead of passed as an (implicit) argument. Unfortunately, the cost of this per-object representation depends on the number of methods the object supports, instead of just the number of fields. In our experience, the extra overhead is bearable when classes are used sparingly, but it becomes overwhelming otherwise.

Finally, the CLOS approach to classes is relatively popular in Scheme, e.g., the Meroon library [26] or Barzilay's Swindle library [7]. In contrast to Smalltalk-style classes, where behaviors are added by changing a class or deriving a new subclass, behavioral extensions in CLOS are attached to *generic methods*. An advantage of this approach is that it provides a clear path for adding "methods" to existing data types, including primitive types like numbers and strings. Another advantage is that it generalizes well to multi-method dispatch, which can easily specialize an operation to a particular combination of classes. A major drawback is that it encourages an imperative programming style, where generic methods are mutated to add new class-specific implementations.

7 Related Work on Mixins and Traits

The terms *mixin* and *trait* have a somewhat troubled and intertwined history, making comparisons among "mixin" and "trait" systems potentially confusing. In this paper, we have committed to particular definitions of the terms, and in the following comparisons, we add a superscript (*, †, or ‡) to each use of a term that does not match our definition.

The term *mixin** originates with Flavors [23], which inspired the Common Lisp Object System (CLOS). In Flavors and CLOS, a mixin* is simply a class that is meant to be combined with other classes via multiple inheritance.

Bracha and Cook refined the definition of *mixin* to "a subclass definition that may be applied to different superclasses" [11]. As defined by Bracha and Cook, mixins subsume classes, and we took a similar approach in our previous model of mixins for Java [17]. Implementations, however, typically define mixins over a base language with classes, as in PLT Scheme and the Jam language [4]. In the same vein, Smaradagkis and

Batory implement mixins with C++ templates [30] in the spirit of our mix of `class` and `lambda`.

For his dissertation, Bracha used the term *mixin*† for a construct in his Jigsaw language [10], which included operations on mixins† such as `sum` and `exclude`. Ancona and Zucca explore a formal framework [5, 6] for mixins†.

Schärli's *traits* [28, 29] are a form of mixin† in the sense of Bracha's dissertation. In particular, Fisher, Reppy, and Turon [14, 27] provide typed models of traits that closely resemble the typed mixin† models of Ancona and Zucca [5, 6]. Using the sense of *mixin* in Bracha and Cook (and PLT Scheme), however, fine-grained operations make traits qualitatively different from mixins. Our encodings of mixins and traits in Scheme illustrate the difference. In practice, Black et al. [9] note the importance of `alias` and `exclude` trait operations for the refactoring of the Smalltalk collection classes. Their experience suggests that mixins are less suited to this kind of refactoring job than traits, but additional experience with both is needed.

The Scala programming language [24] includes a typed `trait`‡ construct, but it does not support any operation on traits‡ other than inheritance and combination with a base class; in other words, the construct may well have been called a *mixin*. Indeed, since multiple Scala traits‡ can be composed when they override the same method, and since the order of the composition determines the resulting pattern of `super` calls, a Scala trait‡ closely resembles a PLT Scheme mixin (but with a static type system). The Fortress [3] language also includes a `trait`‡ construct that is similar to Scala's. Again, Fortress's traits‡ could be characterized as mixins, although the lack of method overriding in Fortress makes the difference nearly insignificant.

Acknowledgements: We thank our PLT colleagues and numerous anonymous users for coping with 11 years of changes to the class system. We wish to acknowledge the financial support of the National Science Foundation and Texas ATP through these years.

References

1. H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1984.
2. N. Adams and J. Rees. Object-oriented programming in Scheme. In *Proc. ACM Conference on Lisp and Functional Programming*, pages 277–288, 1988.
3. E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. The Fortress language specification. 2006.
4. D. Ancona, G. Lagorio, and E. Zucca. Jam - designing a Java extension with mixins. *ACM Transactions on Computing Systems*, 25:641–712, Sept. 2003.
5. D. Ancona and E. Zucca. An algebraic approach to mixins and modularity. In M. Hanus and M. Rodríguez-Artalejo, editors, *Proc. Conference on Algebraic and Logic Programming*, volume 1139 of *Lecture Notes in Computer Science*, pages 179–193. Springer-Verlag, 1996.
6. D. Ancona and E. Zucca. A primitive calculus for module systems. In G. Nadathur, editor, *Proc. International Conference on Principles and Practice of Declarative Programming*, volume 1702 of *Lecture Notes in Computer Science*, pages 62–79. Springer-Verlag, 1999.
7. E. Barzilay. *Swindle*, 2002. `http://www.barzilay.org/Swindle/`.
8. Benson Jr., Brent W. libscheme: Scheme as a C library. In *Proc. USENIX Symposium on Very High Level Languages*, 1994.

9. A. P. Black, N. Schärli, and S. Ducasse. Applying traits to the Smalltalk collection hierarchy. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 47–64, Oct. 2003.
10. G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. Ph.D. thesis, Dept. of Computer Science, University of Utah, Mar. 1992.
11. G. Bracha and W. Cook. Mixin-based inheritance. In *Proc. Joint ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications and the European Conference on Object-Oriented Programming*, Oct. 1990.
12. R. K. Dybvig, R. Hieb, and C. Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, 1993.
13. R. B. Findler, C. Flanagan, M. Flatt, S. Krishnamurthi, and M. Felleisen. DrScheme: A pedagogic programming environment for Scheme. In *Proc. International Symposium on Programming Languages: Implementations, Logics, and Programs*, pages 369–388, Sept. 1997.
14. K. Fisher and J. Reppy. A typed calculus of traits. In *Proc. ACM International Workshop on Foundations of Object-Oriented Languages*, 2004.
15. M. Flatt. Compilable and composable macros. In *Proc. ACM International Conference on Functional Programming*, Oct. 2002.
16. M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *Proc. ACM Conference on Programming Language Design and Implementation*, pages 236–248, June 1998.
17. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 171–183, Jan. 1998.
18. D. P. Friedman. Object-oriented style (invited talk). In *International LISP Conference*, 2003.
19. D. Goldberg, R. B. Findler, and M. Flatt. Super and inner — together at last! In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 116–129, Oct. 2004.
20. R. Kelsey, W. Clinger, and J. Rees (Eds.). The revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9), Sept. 1998.
21. J. Lamping. Typing the specialization interface. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 201–214, 1993.
22. O. Lehrmann Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-oriented programming in the BETA programming language*. ACM Press/Addison-Wesley, 1993.
23. D. A. Moon. Object-oriented programming with Flavors. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–8, Nov. 1986.
24. M. Odersky and M. Zenger. Scalable component abstractions. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 41–57, 2005.
25. *PLT Scheme*, 2006. `www.plt-scheme.org`.
26. C. Queinnec. *Meroon V3: A Small, Efficient, and Enhanced Object System*, 1997.
27. J. Reppy and A. Turon. A foundation for trait-based metaprogramming. In *Proc. ACM International Workshop on Foundations of Object-Oriented Languages*, 2006.
28. N. Schärli. *Composing Classes from Behavioral Building Blocks*. PhD thesis, University of Berne, 2002.
29. N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable units of behaviour. In *Proc. European Conference on Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 248–274. Springer-Verlag, 2003.
30. Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In *Proc. European Conference on Object-Oriented Programming*, pages 550–570, 1998.
31. Smart, J. et al. wxWindows. `http://web.ukonline.co.uk/julian.smart/wxwin/`.