Modular Object-Oriented Programming with Units and Mixins

Robert Bruce Findler Matthew Flatt
Department of Computer Science*
Rice University
Houston, Texas 77005-1892

Abstract

Module and class systems have evolved to meet the demand for reuseable software components. Considerable effort has been invested in developing new module and class systems, and in demonstrating how each promotes code reuse. However, relatively little has been said about the interaction of these constructs, and how using modules and classes *together* can improve programs. In this paper, we demonstrate the synergy of a particular form of modules and classes—called units and mixins, respectively—for solving complex reuse problems in a natural manner.

1 Introduction

Module and class systems both promote code reuse. In theory, many uses of classes can be simulated with modules, and *vice versa*. Experience shows, however, that programmers need both constructs because they serve different purposes [43]. A module delineates boundaries for separate development. A class permits fine-grained reuse via selective inheritance and overriding.

Since modules and classes aid different patterns of reuse, modern languages provide separate constructs for each. Unfortunately, the reuse allowed by conventional module and class systems is limited. In these systems, modules and classes are hard-wired to a specific context, *i.e.*, to specific modules or to a specific superclass.

In previous work, we separately described the novel module and class systems in MzScheme [12]. MzScheme's modules [13], called *units*, are roughly like Java packages, except that units are linked through an external linking specification, instead of through a fixed internal specification. MzScheme's object language [14] provides *mixins*, which are like Java classes except that a mixin is parameterized over its superclass, so it can be applied multiple times to create different derived classes from different base classes. The advantages of units and mixins over conventional module and class languages follow from a single language design principle: *specify connections between modules or classes separately from their definitions.*

The shared principle of separating connections from definitions makes units and mixins synergistic. When units and mixins are combined, a programmer can exploit the encapsulation and linking properties of units to control the application of mixin extensions (*e.g.*, to change the class extended by a particular mixin).

In Section 5, we motivate in more detail the design behind MzScheme's units and mixins, but their synergy is best demonstrated with an example. The bulk of this paper therefore presents an in-depth example, showing how the synergy of units and mixins solves an old extensibility problem [7, 40] in a natural manner. Section 2 describes the extensibility problem, and Section 3 develops a rough solution the problem using conventional classes. Section 4 introduces units and mixins to refine and complete the solution. Sections 5 and 6 extract lessons from the example for the design of modular object-oriented programming languages. Finally, Section 7 relates our work to other research.

2 The Extensibility Problem

The following table summarizes the extensibility problem:

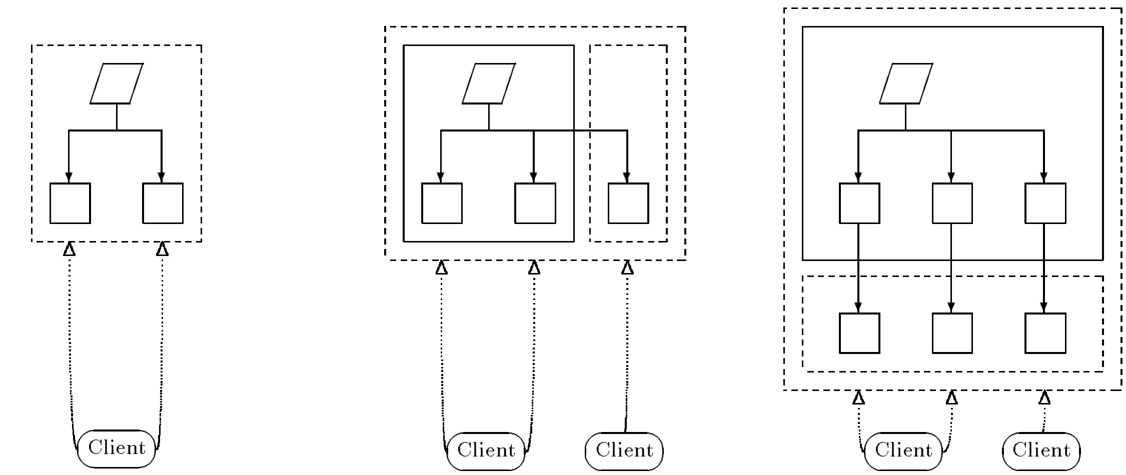
		original variants		extension
		$\Box$	$\bigcirc$	$\leadsto$
original operations	`draw`	`draw(`$\Box$`)`	`draw(`$\bigcirc$`)`	`draw(`$\leadsto$`)`
	`shrink`	`shrink(`$\Box$`)`	`shrink(`$\bigcirc$`)`	`shrink(`$\leadsto$`)`
extension	`rotate`	`rotate(`$\Box$`)`	`rotate(`$\bigcirc$`)`	`rotate(`$\leadsto$`)`

The portion of the table contained in the dotted box represents a program component that provides several operations, `draw` and `shrink`, on a collection of data, geometric shapes like squares and circles. A programmer may wish to use such a component in three different contexts:

1. The programmer may wish to include the component *as is.*
2. The programmer may wish to extend the datatype with a variant, repositioned shapes, and adapt the collection of operations accordingly.
3. The programmer may wish to add a new operation, `rotate`.

*This research was partially supported by a Lodieska Stockbridge Vaughan Fellowship, NSF grants CCR-9619756, CDA-9713032, and CCR-9708957, and a Texas ATP grant.

To appear: ICFP – Sept. 27-29 1998, Baltimore, MD



(a) Original Datatype (b) New Variant (c) New Operation

Figure 1: Extensible programming on datatypes

To avoid duplicate maintenance, or because the component is acquired in object form, the component must be organized so that programmers can add both new forms of data and new operations *without modifying or recompiling*

- the original program component, or
- its existing clients.

Such a program organization dramatically increases the potential for software reuse and enables the smooth integration of proprietary modules.

Neither standard functional nor object-oriented strategies offer a satisfactory way to implement the component and its clients. In a functional language, the variants can be implemented as a type, with the operations as functions on the type. Using this approach, the set of operations is easily extended, but adding a new variant requires modifying the functions. In an object-oriented language, the variants can be implemented as a collection of classes, with the operations as methods common to all of the classes. Using this approach, the datatype is easily extended with a new variant, but adding a new operation is typically implemented by modifying the classes.

The existing literature provides three solutions to the problem. Kühne's [24] solution, which relies on generic procedures with double-dispatching, can interfere with the hierarchical structure of the program. Palsberg and Jay's [32] solution is based on reflection operators and incurs a substantial run-time penalty. Krishnamurthi, Felleisen, and Friedman [11, 23] propose an efficient solution that works with standard class mechanisms, but it requires the implementation (and maintenance) of a complex programming protocol. All of these solutions are partial because they do not address the reuse of clients. In contrast, the combination of units and mixins solves the problem simply and elegantly, and it addresses the reuse of both the original component and its clients.

3 Extensible Programming with Classes

Figure 1 outlines our solution to the extensibility problem:

- Diagram (a) represents the original component. The rhombus stands for the datatype, and the rectangles denote the datatype's variants. The oval is a client of the datatype component.
- Diagram (b) shows the datatype extended with a new variant. The extension is contained in the right inner dashed box. The solid box on the left represents the unmodified datatype code from (a). The original client is also preserved, and a new client of the datatype exploits the variant extension.
- Diagram (c) shows extension in the other direction: adding a new operation to the datatype. As before, the extension is implemented by the inner dashed box while the solid box represents the unmodified existing implementation from (b). The new squares in the extension represent the implementation of the operation for each variant. The existing clients have not been modified, although they now refer to the extended variants.

The remainder of this section develops a concrete example, an evolving shape program [11, 23]. Since Figure 1 can be approximated using conventional classes, we first use only language features available in a typical object-oriented language. But, classes are not enough; Section 4 introduces units and mixins to complete the solution.

3.1 Shape Datatype

Initially, our shape datatype consists of three variants and one operation: rectangles, circles, and translated shapes for drawing. The rectangle and circle variants contain numbers that describe the dimensions of the shape. The translated variant consists of two numbers, Δ_x and Δ_y, and another

```
(define Shape (interface () draw))

(define Rectangle
  (class* null (Shape) (width height)
    (public
      [draw (lambda (window x y) ...)])))

(define Circle
  (class* null (Shape) (radius)
    (public
      [draw (lambda (window x y) ...)])))

(define Translated
  (class* null (Shape) (shape Δx Δy)
    (public
      [draw (lambda (window x y)
              (send shape draw
                    window (+ x Δx) (+ y Δy)))])))
```

Figure 2: Shape classes

```
(define display-shape
  (lambda (shape)
    (if (not (is-a? shape Shape))
      (error "expected a Shape"))
    (let ([window ...])
      (send shape draw window 0 0))))

(display-shape (make-object Translated
                 (make-object Rectangle 50 100)
                 30 30))
```

Figure 3: Two shape clients

shape. For all variants, the drawing operation takes a destination window and two numbers describing a position to draw the shape.

The shape datatype is defined by the Shape interface and implemented by three classes: Rectangle, Circle and Translated. Each subclass declares a *draw* method, which is required to implement the Shape interface. Figure 2 shows the interface and class definitions using MzScheme's class system. (MzScheme's class system is similar to Java's; for details, see the Appendix.)

Figure 3 contains clients for the shape datatype. The function *display-shape* consumes a shape and draws it in a new window. The final expression creates a shape and displays it. As the shape datatype is extended, we consider how these clients are affected.

```
(define Union
  (class* null (Shape) (left right)
    (public
      [draw (lambda (window x y)
              (send left draw window x y)
              (send right draw window x y))])))

(display-shape
  (make-object Union
    (make-object Rectangle 10 30)
    (make-object Translated
      (make-object Circle 20) 30 30)))
```

Figure 4: Variant extension and a new client

```
(define BB-Shape (interface (Shape) bounding-box))

(define BB-Rectangle
  (class* Rectangle (BB-Shape) (width height)
    (public
      [bounding-box
       (lambda () (make-object BB 0 0 width height))])
    (sequence (super-init width height))))

(define BB-Circle
  (class* Circle (BB-Shape) (radius)
    (public
      [bounding-box
        (lambda () (make-object BB (- radius) (- radius)
                                   radius radius))])
    (sequence (super-init r))))

(define BB-Translated
  (class* Translated (BB-Shape) (shape Δx Δy)
    (public
      [bounding-box (lambda () ...)])
    (sequence (super-init shape Δx Δy))))

(define BB-Union
  (class* Union (BB-Shape) (left right)
    (public
      [bounding-box (lambda () ...)])
    (sequence (super-init left right))))

(define BB
  (class* null () (left top right bottom)
    ...))

(define display-shape
  (lambda (shape)
    (if (not (is-a? shape BB-Shape))
      (error "expected a BB-Shape"))
    (let* ([bb (send shape bounding-box)]
           [window ...] [x ...] [y ...])
      (send shape draw window x y))))
```

Figure 5: Operation extension

3.2 Variant Extension

To create more interesting configurations of shapes, we extend the shape datatype with a new variant representing the union of two shapes. Following the strategy suggested in Figure 1 (b), we define a new Union class derived from Shape. Figure 4 defines the Union class, and shows an expression that uses the new class.

The simplicity of the variant extension reflects the natural expressiveness of object-oriented programming. The object-oriented approach also lets us add this variant without modifying the original code or the existing clients in Figure 3.

3.3 Operation Extension

Shapes look better when they are drawn centered in their windows. We can support centered shapes by adding the operation *bounding-box*, which computes the smallest rectangle enclosing a shape.

We add an operation to our shape datatype by defining four new classes, each derived from the variants of Shape in Section 3.2. Figure 5 defines the extended classes BB-Circle, BB-Rectangle, BB-Translated, and BB-Union, each providing the *bounding-box* method. It also defines the BB-Shape

```
(set! factory ...)
...
(display-shape (send factory make-union
                 (send factory make-rectangle 10 30)
                 (send factory make-translated
                   (send factory make-circle 20) 30 30)))
```

Figure 6: Revised clients using Abstract Factory

interface, which describes the extended shape type for the bounding box classes just as Shape describes the type for the original shape classes.

The new *display-shape* client in Figure 5 uses bounding box information to center its shape in a window. Unfortunately, we must also modify the clients so they create instances of the new bounding box classes instead of the original shape classes, including clients that do not use bounding box information. Thus, the standard object-oriented architecture does not satisfy our original goal; it does not support operation extensions to the shape datatype without modifying existing clients.

Since object-oriented programming constructs do not address this problem directly, we must resort to a programming protocol or pattern. In this case, the Abstract Factory pattern [15] and a mutable reference solves the problem. The Abstract Factory pattern relies on one object, called the *factory*, to create instances of the shape classes. The factory supplies one creation method for each variant of the shape, and clients create shapes by calling these methods instead of using make-object directly. To change the classes that are instantiated by clients, it is only necessary to change the factory, which is stored in the mutable reference. A revised client, using the Abstract Factory, is shown in Figure 6.

The Abstract Factory pattern implements a simple dynamic linker, where **set!** installs the link.[1] It separates the definition of shapes and clients so that a specific shape implementation can be selected at a later time, rather than hard-wiring a reference to a particular implementation into the client. However, using a construct like **set!** for linking obscures this intent both to other programmers and to the compiler. A more robust solution is to improve the module language.

4 Better Reuse through Units and Mixins

In the previous section, we developed the Shape datatype and its collection of operations, and we showed how object-oriented programming supports new variants and operations in separately developed extensions. In this section, we make the separate development explicit using MzScheme's module system; the basic definitions, the extensions, and the clients are all defined in separate modules. MzScheme supports separate compilation for these modules, and provides a flexible language for linking them. Indeed, the linking implemented with an Abstract Factory in the previous section can be more naturally defined through module linking. Finally, we show how MzScheme's class-module combination provides new opportunities for reuse that are not available in conventional object-oriented languages.

[1] Factory Method is a related pattern where an extra operation in the datatype is used to create instances instead of a separate factory object. Factory Method applies to an interesting special case: the datatype client and the datatype implementation are the same, thus making the datatype implementation extensible.

```
(define BASIC-SHAPES
  (unit (import)
        (export Shape Rectangle Circle Translated)
    (define Shape (interface ...))  ; see Figure 2
    (define Rectangle (class* null (Shape) ...))
    (define Circle (class* null (Shape) ...))
    (define Translated (class* null (Shape) ...))))
```

Figure 7: Creating Units

```
(define GUI
  (unit (import Shape)
        (export display-shape)
   (define display-shape ...))) ; see Figure 3

(define PICTURE
  (unit (import Rectangle Circle Translated display-shape)
        (export)
   (display-shape (make-object ...)))) ; see Figure 3
```

Figure 8: Unitized shape clients

4.1 Unitizing the Basic Shapes

A module in MzScheme is called a *unit*. Figure 7 shows the basic shape classes encapsulated in a BASIC-SHAPES unit. This unit imports nothing and exports all of the basic shape classes. The body of the unit contains the class definitions exactly as they appear in Figure 2.

In general, the shape of a unit expression is

$$\begin{array}{l}(\textbf{unit}\ (\textbf{import}\ \textit{variable}\ \cdots) \\ \quad\quad (\textbf{export}\ \textit{variable}\ \cdots) \\ \quad \textit{unit-body-expr}\ \cdots)\end{array}$$

(centered ellipses indicate repeated syntactic patterns). The *unit-body-exprs* have the same form as top-level Scheme expressions, allowing a mixture of expressions and definitions, but **define** within a **unit** expression creates a unit-local variable instead of a top-level variable. The unit's imported variables are bound within the *unit-body-exprs*. Each exported variable must be defined by some *unit-body-expr*. Unexported variables that are defined in the *unit-body-exprs* are private to the unit.

Figure 8 defines two client units of BASIC-SHAPES: GUI and PICTURE. The GUI unit provides the function *display-shape* (the same as in Figure 3). Since it only depends on the functionality in the Shape type, not the specific variants, it only imports Shape. The PICTURE unit imports all of the shape variants—so it can construct instances—as well as the *display-shape* function, and it exports nothing. When PICTURE is invoked as part of a program, it constructs a shape and displays it.

A unit is an unevaluated bundle of code, much like a ".o" object file created by compiling a traditional language. At the point where BASIC-SHAPES, GUI, and PICTURE are defined as units, no shape classes have been defined, no instances have been created, and no drawing window has been opened. Each unit encapsulates its definitions and expressions without evaluating them until the unit is invoked, just like a procedure encapsulates expressions until it is applied. However, none of the units in Figures 7 and 8 can be invoked directly because each unit requires imports. The units must first be linked together to form a program.

```
(define BASIC-PROGRAM
 (compound-unit
   (import)
   (link [S (BASIC-SHAPES)]
         [G (GUI (S Shape))]
         [P (PICTURE (S Rectangle) (S Circle) (S Translated)
                     (G display-shape))])
   (export)))

(invoke-unit BASIC-PROGRAM)
```

Figure 9: Linking basic shape program

4.2 Linking the Shape and Client Units

Units are linked together with the **compound-unit** form. Figure 9 shows how to link the units of the previous subsection into a complete program: BASIC-PROGRAM. The PICTURE unit's imports are not *a priori* associated with the classes in BASIC-SHAPES. This association is established only by the **compound-unit** expression, and it is established only in the context of BASIC-PROGRAM. The PICTURE unit can be reused with different Shape classes in other compound units.

The **compound-unit** form links several units, called *constituent* units, into one new *compound* unit. The linking process matches imported variables in each constituent unit with either variables exported by other constituents, or variables imported into the compound unit. The compound unit can then re-export some of the variables exported by the constituents. Thus, BASIC-PROGRAM is a unit with imports and exports, just like BASIC-SHAPES or GUI, and no evaluation of the unit bodies has occurred. But, unlike the BASIC-SHAPES and GUI units, BASIC-PROGRAM is a complete program because it has no imports.

Each **compound-unit** expression

```
(compound-unit (import variable ···)
               (link [tag_1 (expr_1 linkspec_1 ···)]
                       ⋮
                     [tag_n (expr_n linkspec_n ···)])
               (export (tag variable) ···))
```

has three parts:

- The **import** clause lists variables that are imported into the compound unit. These imported variables can be linked to the constituent unit's imports.
- The **link** clause specifies how the compound unit is created from the constituent units. Each constituent unit is specified via an *expr* and identified with a unique *tag*. Following the *expr*, a link specification *linkspec* is provided for each of the constituent's imports. Link specifications have two forms:
  - A *linkspec* of the form *variable* links the constituent's import to the corresponding import of the compound unit.
  - A *linkspec* of the form (*tag variable*) links the constituent's import to *variable* as exported by the *tag* constituent.
- The **export** clause re-exports variables from the compound unit that are exported from the constituents. The *tag* indicates the constituent and *variable* is the variable exported by that constituent.

```
(define UNION-SHAPE
  (unit (import Shape)
        (export Union)
    (define Union (class* null (Shape) ...)))) ; see Figure 4

(define BASIC+UNION-SHAPES
  (compound-unit
    (import)
    (link [S (BASIC-SHAPES)]
          [US (UNION-SHAPE (S Shape))])
    (export (S Shape)
            (S Rectangle)
            (S Circle)
            (S Translated)
            (US Union))))
```

Figure 10: Variant extension in a unit

To evaluate a **compound-unit** expression, the *expr*s in the **link** clause are evaluated to determine the compound unit's constituents. For each constituent, the number of variables it imports must match the number of *linkspec*s provided; otherwise, an exception is raised. Each *linkspec* is matched to an imported variable by position.[2] Each constituent must also export the variables that are referenced by **link** and **export** clauses using the constituent's *tag*.

Once a compound unit's constituents are linked, the compound unit is indistinguishable from an atomic unit. Conceptually, linking creates a new unit by merging the internal expressions and definitions from all the constituent units. During this merge, variables are renamed as necessary to implement linking between constituents and to avoid name collisions between unrelated variables. The merged *unit-body-expr*s are ordered to match the order of the constituents in the **compound-unit**'s **link** clause.[3]

4.3 Invoking Unit Programs

The BASIC-PROGRAM unit from Figure 9 is a complete program, analogous to a conventional application, but the program still has not been executed. In most languages with module systems, a complete program is executed through commands outside the language. In MzScheme, a program unit is executed directly with the **invoke-unit** form:

(**invoke-unit** *expr*)

The value of *expr* must be a unit. Invocation evaluates the unit's definitions and expressions, and the result of the last expression in the unit is the result of the **invoke-unit** expression. Hence, to run BASIC-PROGRAM, evaluate

(**invoke-unit** BASIC-PROGRAM)

4.4 New Units for a New Variant

To extend Shape with a Union variant, we define the extension in its own unit, UNION-SHAPE, as shown in Figure 10.

[2] In MzScheme's extended unit language with signatures, linking matches variables by name rather than by position. When the number of imports is small, linking by position is simpler because it avoids complex machinery for renaming variables.

[3] The implementation of linking is equivalent to this reduction, but far more efficient. In particular, it is not necessary to extract expressions from the constituent units, which would break separate compilation.

```
(define UNION-PICTURE
  (unit (import Rectangle Circle Translated Union
                display-shape)
        (export)
    (display-shape (make-object ...)))) ; see Figure 4

(define UNION-PROGRAM
  (compound-unit
   (import)
   (link [S (BASIC+UNION-SHAPES)]
         [G (GUI (S Shape))]
         [P (PICTURE (S Rectangle) (S Circle) (S Translated)
                     (G display-shape))]
         [UP (UNION-PICTURE (S Rectangle)
                            (S Circle)
                            (S Translated)
                            (S Union)
                            (G display-shape))])
   (export)))

(invoke-unit UNION-PROGRAM)
```

Figure 11: New client and the extended program

The Shape class is imported into UNION-SHAPE, and the new Union class is exported. In terms of Figure 1 (b), UNION-SHAPE corresponds to the smaller dashed box, drawn around the new variant class. The solid box is the original unmodified BASIC-SHAPES unit, and the outer dashed box in (b) is BASIC+UNION-SHAPES, a compound unit linking UNION-SHAPE together with BASIC-SHAPES.

Since the BASIC+UNION-SHAPES unit exports the variants defined by both BASIC-SHAPES and UNION-SHAPE, it can serve as a replacement for the original BASIC-SHAPES unit, yet it can also provide more functionality for new clients. The UNION-PROGRAM unit in Figure 11 demonstrates both of these uses. In this new program, the GUI and PICTURE clients are reused intact from the original program, but they are now linked to BASIC+UNION-SHAPES instead of BASIC-SHAPES. An additional client unit, UNION-PICTURE, takes advantage of the shape extension to draw a superimposed rectangle and circle picture.

4.5 New Units and Mixins for a New Operation

To extend Shape with a *bounding-box* operation, we define the BB-SHAPES unit in Figure 12. This unit corresponds to the smaller dashed box in Figure 1 (c).

The BB-SHAPES unit is the first example to rely on mixins. The BB-Rectangle class is derived from an imported Rectangle class, which is not determined until the unit is linked—long after the unit is compiled. Thus, BB-Rectangle defines a *mixin*, a class extension that is parameterized over its superclass.

The compound unit BASIC+UNION+BB-SHAPES links the BASIC+UNION-SHAPES unit from the previous section with the new bounding-box unit, then exports the bounding-box classes. As the bounding-box classes are exported, they are renamed to match the original class names,[4] *i.e.*, BB-Rectangle is renamed to Rectangle, and so on. This renaming does not affect the linking *within* BASIC+UNION+BB-SHAPES; it only affects the way that BASIC+UNION+BB-SHAPES is linked with other units.

[4]The simplified description of **compound-unit** in Section 4.2 did not cover the syntax for renaming exports. For a complete description of **compound-unit**, see the MzScheme manual [12].

```
(define BB-SHAPES
  (unit (import Shape Rectangle Circle Translated Union)
        (export BB-Shape BB-Rectangle BB-Circle
                BB-Translated BB-Union BB)
    (define BB-Shape (interface (Shape) ...)) ; see Figure 5
    (define BB-Rectangle (class* Rectangle ...))
    (define BB-Circle (class* Circle ...))
    (define BB-Translated (class* Translated ...))
    (define BB-Union (class* Union ...))
    (define BB ...)))

(define BASIC+UNION+BB-SHAPES
  (compound-unit
   (import)
   (link [S (BASIC+UNION-SHAPES)]
         [BS (BB-SHAPES (S Shape)
                        (S Rectangle)
                        (S Circle)
                        (S Translated)
                        (S Union))])
   (export (S Shape)
           (BS BB-Shape) (BS BB)
           ; rename BS's BB-Rectangle to Rectangle, etc.:
           (BS (BB-Rectangle Rectangle))
           (BS (BB-Circle Circle))
           (BS (BB-Translated Translated))
           (BS (BB-Union Union)))))
```

Figure 12: Operation extension in a unit

```
(define BB-GUI
  (unit (import BB-Shape BB)
        (export display-shape)
    (define display-shape
      (lambda (shape)
        (if (not (is-a? shape BB-Shape))
            ...) ; see Figure 5
        ...))))

(define BB-PROGRAM
 (compound-unit
  (import)
  (link [S (BASIC+UNION+BB-SHAPES)]
        [BG (BB-GUI (S BB-Shape) (S BB))]
        [P (PICTURE (S Rectangle) (S Circle) (S Translated)
                    (BG display-shape))]
        [UP (UNION-PICTURE (S Rectangle) (S Circle)
                           (S Translated) (S Union)
                           (BG display-shape))])
  (export)))

(invoke-unit BB-PROGRAM)
```

Figure 13: Program with the operation extension

As before, the BASIC+UNION+BB-SHAPES unit serves as a replacement for either BASIC-SHAPES or BASIC+UNION-SHAPES, and also provides new functionality for new clients. One new client is BB-GUI (see Figure 13), which provides a *display-shape* that exploits bounding box information to center a shape in a window. The BB-GUI unit replaces GUI, but we reuse PICTURE and UNION-PICTURE without modifying them. An Abstract Factory is unnecessary because units already permit us to vary the connection between the shape-creating clients and the instantiated classes. Putting everything together produces the new program BB-PROGRAM at the bottom of Figure 13.

```
(define COLOR-SHAPE
  (unit (import Shape)
        (export C-Shape)
      (define C-Shape
        (class* Shape () args
          (rename
            [super-draw draw])
          (public
            [color "black"]
            [change-color
              (lambda (c)
                (set! color c))]
            [draw
              (lambda (window x y)
                (send window set-color color)
                (super-draw window x y))])
          (sequence
            (apply super-init args))))))

(define BASIC+UNION+BB+COLOR-SHAPES
  (compound-unit
   (import)
   (link [S (BASIC+UNION+BB-SHAPES)]
         [CR (COLOR-SHAPE (S Rectangle))]
         [CC (COLOR-SHAPE (S Circle))]
         [CT (COLOR-SHAPE (S Translated))]
         [CU (COLOR-SHAPE (S Union))])
   (export (S Shape)
           (S BB-Shape)
           (S BB)
           (CR (C-Shape Rectangle))
           (CC (C-Shape Circle))
           (CT (C-Shape Translated))
           (CU (C-Shape Union)))))
```

Figure 14: Reusing a class extension

4.6 Synergy at Work

The shape example demonstrates the synergy of units and mixins. Units, by separating the definition and linking of modules, support the reuse of PICTURE and UNION-PICTURE as the shape representation evolves. Mixins, by abstracting a class expression over an imported class, enable the encapsulation of each extension in its own unit. The combination of units and mixins thus enables a direct translation of the ideal program structure from Figure 1 into a working program.

We have achieved the complete reuse of existing code at every stage in the extension of Shape, but even *more* reuse is possible. The code in Figure 14 illustrates how units and mixins combine to allow the use of one *extension* multiple times. The COLOR-SHAPE unit imports a Shape class and extends it to handle colors. With this single unit containing a single mixin, we can extend all four of the shape variants: Rectangle, Circle, Translated, and Union. The compound unit BASIC+UNION+BB+COLOR-SHAPES in Figure 14 uses the COLOR-SHAPE unit four times to obtain the set of color shape classes.

The code in Figure 14 uses a few features that are not described in this paper (the **rename** clause in a **class*** expression, and the use of *args* to stand for multiple arguments, passed on to super-init with apply). These details are covered in the MzScheme reference manual [12]. The point here is that units and mixins open new avenues for reuse on a large scale.

5 The Moral of the Story

We have demonstrated how units and mixins apply to a specific example, but a general principle is at work: specifying connections between modules or classes separately from their definitions. This principle is the key to making units and mixins succeed together without conflating the distinct purposes of module and class systems.

A module system serves two key purposes:

- **Separate Development:** A module encapsulates a set of definitions, clearly delineating the interface between the module and the rest of the program. Each module can be developed in isolation and distributed to clients in a compiled form.
- **Linking:** Modules are linked together to form a program. Linking connects the definitions in one module with those in another, but a module cannot interfere with the internal structure of any other module.

In contrast, a class system supports three different key services:

- **Extensible Types:** An interface defines an extensible type, and a class implements such a type.
- **Selective Reuse:** A class can selectively refine the implementation of its superclass, preserving some inherited definitions and overriding others with its own definitions.
- **Instantiation:** A class is instantiated to create an object, a first-class value that encapsulates the methods and instance variables of the class.

To serve their distinct purposes, modules and classes require distinct constructs in a programming language, but these constructs interact. In our example program, the collection of geometric shapes is naturally implemented as a set of Shape classes. The implementation of the shape classes and the client code that uses them are defined in separate modules. Using classes to represent shapes makes it easy to extend the shape classes without modifying the basic definition of a shape. Separating the definition of shapes from their use in different modules makes it easy to replace the original shape classes with new classes without modifying the client. This is precisely how modular object-oriented code is supposed to work.

Unfortunately, the existing module-class combinations do not support this sort of modular object-oriented programming. In Java, for example, if the Rectangle class is extended, then a client module that creates Rectangle instances must be modified to refer to the new, extended class. The root of this problem, both in Java and many other object-oriented languages, is that the connections between modules are hard-wired within the modules. For example, client modules declare that they import *the* shape module instead of importing *a* shape module.

The design of module and class constructs must encourage the interaction of the constructs. The Shape example suggests a lesson for the design of modules:

> *Separate a module's linking specification from its encapsulated definitions.*

In other words, a module should describe its imports with enough detail to support separate compilation, but the module should not specify the source of its imports. Instead, the imports should be supplied externally by a linking expression.

A module system with external linking in turn constrains the design of the class system. A module may encapsulate a class definition with an imported superclass (*e.g.*, BB-Rectangle in Figure 12). Since module linking is specified outside the module, the superclass is not determined until the module is linked, so the class expression is *de facto* parameterized over its superclass. Such a parameterized class is a *mixin*. Mixins tend to be computationally more expensive than classes, but the cost is small [14]. In parallel to the lesson for modules, the requirement to support mixins can be stated as follows:

> *Separate a class's superclass specification from its extending definitions.*

Mixins are valuable in their own right. While classes enable reuse because each class can be extended and refined by defining new subclasses, the reuse is one-sided: each class can be extended in many different ways, but each extension applies only to its superclass. A mixin is parameterized with respect to its superclass, so it can add functionality to many different classes. Thus, the reuse potential of a mixin is greater than that of a class.

6 A Type Challenge

We have explored typed models of mixins [14] and units [13] separately in previous work. In addition, we have anticipated an extension of the present work with types by including is-a? safety tests in our examples, and by showing how the Shape and BB-Shape interfaces are linked to clients to enable those tests. Still, certain challenges remain for bringing mixins and units together in a single typed model. For mixins, the previously published type rules assume a complete program and a single namespace for mixin names. For units, the previously published language does not express the kind of type relationships necessary for importing and exporting interface types (*e.g.*, importing types A and B where A must be a subtype of B).

Others have explored a similar combination of classes and modules in a typed setting. The module systems in Objective Caml [28, 38] and OML [39] support externally specified connections, and since a class can be defined within a module, these languages also provide a form of mixins. However, the modules and mixins in these languages fail to support the synergy we have demonstrated for units and mixins. In particular, they do not allow the operation extension demonstrated in Section 4 because an imported class must match the expected type exactly—no extra methods are allowed. In our example, PICTURE is initially linked to the Rectangle class and later linked to BB-Rectangle; since the latter has more methods, neither Objective Caml nor OML would allow PICTURE to be reused in this way. Our example thus suggests a third lesson for the design of module and class type systems:

> *Allow subsumption for connections, including both module linking and class extension.*

7 Related Work

Much of the previous research on modules and classes focused on unifying the constructs. Lee and Friedman [26, 27] investigated languages that work directly on variables and bindings, which provides a theoretical foundation for implementing both modules and classes. Similarly, Jagannathan [21] and Miller and Rozas [29] proposed first-class environments as a common mechanism. Bracha [3] has explored mixins for both modular and object-oriented programming; Ancona and Zucca [1] provide a categorical treatment of this view. Our work is complementary to all of the above work, because we focus on designing the constructs to be used by a programmer, rather than the method used to implement those constructs.

Languages that have promoted modularization, including Mesa [31], Modula-2 [45], and SML [30], provide no direct support for object-oriented programming. Similarly, early object-oriented languages, such as Simula 67 [9] and Smalltalk [16], provide no module system. In contrast, languages such as Ada 95 [20], Common Lisp [42], Dylan [41], Haskell [19], Java [17], and Modula-3 [18] provide both modules and classes. For Cecil [4], Chambers and Leavens [5] designed a module system specifically to complement a class system with multi-methods. Unfortunately, these module and class systems do not support external connections—a central principle of our design that is crucial to software engineering (see Section 5).

Scheme [6] provides no specific mechanisms for modular or object-oriented programming. Instead, Scheme supports modular programming through lexical scope, and many implementations provide separate compilation for top-level expressions. Programmers can regard top-level expressions as "modules" that hide private definitions by using **let** or **letrec**. A number of Scheme systems have been developed that codify the module-via-top-level idea [8, 10, 36, 35, 44], but none of these satisfies the criteria in Section 5. In contrast, Kelsey's proposed module system [22] captures many of the same ideas as units. Scheme can also support object-oriented programming by simulating objects with procedures and classes with higher-order procedures [37]. Several object-oriented extensions for Scheme have been developed [2, 34], including some that support mixins [25, 33].[5] However, none of these systems provide complete languages for both modular and object-oriented programming.

8 Conclusion

Units and mixins promote a synergistic integration of modular and object-oriented programming techniques. The combination succeeds due to a consistent separation of definitions (encapsulated in modules or classes) from connections (between modules or classes) in both units and mixins.

The bulk of the paper explores the extensibility problem because it highlights many of the advantages of units and mixins. Strictly speaking, the problem can be solved using conventional module and class systems and the Abstract Factory pattern. Nevertheless, a straightforward datatype implementation using units and mixins is more immediately extensible. This natural bias towards reuse and extension is the essential benefit of units and mixins.

For a complete version of the code presented here, see

`www.cs.rice.edu/CS/PLT/Publications/#ifcp98-ff`

[5] Queinnec's [33] system actually provides generic function extensions that are parameterized over a generic function, rather than parameterized class extensions. While the system does not provide mixins *per se*, it follows the principle of separating connections from definitions.

Acknowledgements

The authors would like to thank Matthias Felleisen, Corky Cartwright, John Clements, Dan Friedman, Shriram Krishnamurthi, Paul Steckler, and the anonymous reviewers.

References

[1] Ancona, D. and E. Zucca. An algebraic approach to mixins and modularity. In Hanus, M. and M. Rodríguez-Artalejo, editors, *Proc. Conference on Algebraic and Logic Programming*, Lecture Notes in Computer Science 1139, pages 179–193, Berlin, 1996. Springer Verlag.

[2] Bartley, D. H. and J. C. Jensen. The implementation of PC Scheme. In *Proc. ACM Conference on Lisp and Functional Programming*, pages 86–93, 1986.

[3] Bracha, G. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance.* Ph.D. thesis, Dept. of Computer Science, University of Utah, March 1992.

[4] Chambers, C. *The Cecil Language Specification and Rationale: Version 2.0*, 1995.

[5] Chambers, C. and G. T. Leavens. Typechecking and modules for multi-methods. *ACM Transactions on Programming Languages and Systems*, 17(6):805–843, November 1995.

[6] Clinger, W. and Rees, J. (Eds.). The revised^4 report on the algorithmic language Scheme. *ACM Lisp Pointers*, 4(3), July 1991.

[7] Cook, W. R. Object-oriented programming versus abstract data types. In *Foundations of Object-Oriented Languages*, pages 151–178, June 1990.

[8] Curtis, P. and J. Rauen. A module system for Scheme. In *Proc. ACM Conference on Lisp and Functional Programming*, pages 13–28, 1990.

[9] Dahl, O.-J., B. Myrhaug and K. Nygaard. SIMULA 67. common base language. Technical Report Publ. No. S-2, Norwegian Computing Center, Oslo, Norway, May 1968.

[10] Feeley, M. *Gambit-C, a portable Scheme implementation*, 1996.

[11] Felleisen, M. and D. P. Friedman. *A Little Java, A Few Patterns.* The MIT Press, 1998.

[12] Flatt, M. PLT MzScheme: Language manual. Technical Report TR97-280, Rice University, 1997.

[13] Flatt, M. and M. Felleisen. Units: Cool modules for HOT languages. In *Proc. ACM Conference on Programming Language Design and Implementation*, pages 236–248, 1998.

[14] Flatt, M., S. Krishnamurthi and M. Felleisen. Classes and mixins. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 171–183, 1998.

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

[16] Goldberg, A. and D. Robson. *Smalltalk 80: The Language.* Addison-Wesley, Reading, 1989.

[17] Gosling, J., B. Joy and G. Steele. *The Java Language Specification.* The Java Series. Addison-Wesley, Reading, MA, USA, June 1996.

[18] Harbison, S. P. *Modula-3.* Prentice Hall, 1991.

[19] Hudak, P. and Wadler, P. (Eds.). Report on the programming language Haskell. Technical Report YALE/DCS/RR777, Yale University, Department of Computer Science, August 1991.

[20] International Organization for Standardization. *Ada 95 Reference Manual. The Language. The Standard Libraries*, Janurary 1995.

[21] Jagannathan, S. Metalevel building blocks for modular systems. *ACM Transactions on Programming Languages and Systems*, 16(3):456–492, May 1994.

[22] Kelsey, R. A. Fully-parameterized modules or the missing link. Technical Report 97-3, NEC Research Institute, 1997.

[23] Krishnamurthi, S., M. Felleisen and D. Friedman. Synthesizing object-oriented and functional design to promote re-use. In *Proc. European Conference on Object-Oriented Programming*, 1998.

[24] Kühne, T. The translator pattern—external functionality with homomorphic mappings. In *Proceedings of TOOLS 23, USA*, pages 48–62, July 1997.

[25] Lang, K. J. and B. A. Pearlmutter. Oaklisp: an object-oriented dialect of Scheme. *Lisp and Symbolic Computation: An International Journal*, 1(1):39–51, May 1988.

[26] Lee, S.-D. and D. P. Friedman. Quasi-static scoping: Sharing variable bindings across multiple lexical scopes. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 479–492, 1993.

[27] Lee, S.-D. and D. P. Friedman. Enriching the lambda calculus with context toward a theory of incremental program construction. In *Proc. ACM International Conference on Functional Programming*, pages 239–250, 1996.

[28] Leroy, X. *The Objective Caml system*, 1996. URL: `http://pauillac.inria.fr/ocaml/`.

[29] Miller, J. and G. Rozas. Free variables and first-class environments. *Lisp and Symbolic Computation: An International Journal*, 3(4):107–141, 1991.

[30] Milner, R., M. Tofte and R. Harper. *The Definition of Standard ML.* The MIT Press, Cambridge, Massachusetts and London, England, 1990.

[31] Mitchell, J. G., W. Mayberry and R. Sweet. *Mesa Language Manual*, 1979.

[32] Palsberg, J. and C. B. Jay. The essence of the Visitor pattern. Technical Report 05, University of Technology, Sydney, 1997.

[33] Queinnec, C. Distributed generic functions. In *Proc. 1997 France-Japan Workshop on Object-Based Parallel and Distributed Computing*, 1997.

[34] Queinnec, C. *Meroon V3: A Small, Efficient, and Enhanced Object System*, 1997.

[35] Queinnec, C. and D. De Roure. Sharing code through first-class envinroments. In *Proc. ACM International Conference on Functional Programming*, pages 251–261, 1996.

[36] Rees, J. Another module system for Scheme, 1994. Scheme48 documentation.

[37] Rees, J. and N. Adams. Object-oriented programming in Scheme. In *Proc. ACM Conference on Lisp and Functional Programming*, pages 277–288, 1988.

[38] Rémy, D. and J. Vouillon. Objective ML: A simple object-oriented extension of ML. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 40–53, Paris, France, 15–17 Janurary 1997.

[39] Reppy, J. and J. Riecke. Simple objects for Standard ML. In *Proc. ACM Conference on Programming Language Design and Implementation*, pages 171–180, 1996.

[40] Reynolds, J. C. User-defined types and procedural data structures as complementary approaches to data abstraction. In Schuman, S. A., editor, *New Directions in Algorithmic Languages*, pages 157–168. IFIP Working Group 2.1 on Algol, 1975.

[41] Shalit, A. *The Dylan Reference Manual*. Addison-Wesley, 1996.

[42] Steele Jr., G. L. *Common Lisp: The Language*. Digital Press, second edition, 1990.

[43] Szyperski, C. A. Import is not inheritance – why we need both: Modules and classes. In *European Conference on Object-Oriented Programming*, volume 615 of *Lecture Notes in Computer Science*, pages 19–32. Springer-Verlag, New York, N.Y., 1992.

[44] Tung, S.-H. Interactive modular programming in Scheme. In *Proc. ACM Conference on Lisp and Functional Programming*, pages 86–95, 1992.

[45] Wirth, N. *Programming in Modula-2*. Springer-Verlag, 1983.

Appendix: MzScheme Class and Interface Syntax

Classes

The shape of a MzScheme class declaration is:

(**class***
  superclass-expr (*interface-expr* ···) (*init-variable* ···)
  instance-variable-clause ···)

(centered ellipses indicate repeated patterns). The expression *superclass-expr* determines the superclass for the new class, and the *interface-expr*s specify the interfaces implemented by the class. The *init-variable*s receive instance-specific initialization values when the class is instantiated (like the arguments supplied with **new** in Java). Finally, the *instance-variable-clause*s define the instance variables of the class, plus expressions to be evaluated for each instance. For example, a **public** clause declares public instance variables and methods.

Thus, the definition

```
(define Rectangle
  (class* null (Shape) (width height)
    (public
      [draw (lambda (window x y) ···)])))
```

introduces the base class Rectangle. The null indicates that Rectangle is not derived from any class, (Shape) indicates that it implements the Shape interface, and (*width height*) indicates that two initialization arguments are consumed for initializing an instance. There is one *instance-variable-clause* that defines a public method: *draw*.

MzScheme's object system does not distinguish between instance variables and methods. Instead, procedure-valued instance variables act like methods. The *draw* declaration in Rectangle defines an instance variable, and (**lambda** (*window x y*) ...) is its initial value expression, evaluated once per instance. When *draw* is called as the method of some object, *draw* may refer to the object via *this*. In most object-oriented languages, *this* is passed in as an implicit argument to a method; in MzScheme, *this* is part of the environment for evaluating initialization expression, so each "method" in an object is a closure containing the correct value of *this*.[6]

An instance of Rectangle is created using the make-object primitive. Along with the class to instantiate, make-object takes any initialization arguments that are expected for the class. In the case of Rectangle, two initialization arguments specify the size of the shape:

```
(define rect (make-object Rectangle 50 100))
```

The value of an instance variable is extracted from an object using **ivar**. The following expression calls the *draw* "method" of *rect* by extracting the value of *draw* and applying it as a procedure:

```
((ivar rect draw) window 0 0)
```

Since method calls of this form are common, MzScheme provides a **send** macro. The following **send** expression is equivalent to the above **ivar** expression:

```
(send rect draw window 0 0)
```

[6] MzScheme's approach to methods avoids duplicating the functionality of procedures with methods. However, this design incurs a substantial cost in practice because each object record must provide a slot for every method in the class, and a closure is created for each method per object. Adding true methods to the object system, like methods in most object-oriented languages, would improve the run-time performance of the object system and would not affect the essence of our presentation.

Interfaces

An interface is declared in MzScheme using the **interface** form:

```
(interface (superinterface-expr ···)
  variable ···)
```

The *superinterface-expr*s specify all of the superinterfaces for the new interface, and the *variable*s are the instance variables required by the interface (in addition to variables declared by the superinterfaces). For example, the definition

```
(define Shape (interface () draw))
```

creates an interface named Shape with one variable: *draw*. Every class that implements Shape must declare a *draw* instance variable. The definition

```
(define BB-Shape (interface (Shape) bounding-box))
```

creates an interface named BB-Shape with two variables: *draw* and *bounding-box*. Since Shape is the superinterface of BB-Shape, every class that implements BB-Shape also implements Shape.

A class implements an interface only when it specifically declares the implementation (as in Java). Thus, the Rectangle class in the previous section only implements the Shape interface.

Derived Classes

The definition

```
(define BB-Rectangle
  (class* Rectangle (BB-Shape) (width height)
    (public [bounding-box ···])
    (sequence (super-init width height))))
```

derives a BB-Rectangle class that implements BB-Shape. The *draw* method, required to implement BB-Shape, is inherited from Rectangle.

The BB-Rectangle class declares the new *bounding-box* method. It also includes a **sequence** clause that calls super-init. A **sequence** clause declares expressions to be evaluated for each instance. It is commonly used to call the special super-init procedure, which initializes the part of the instance defined by the superclass (like calling **super** in a Java constructor); a derived class must call super-init exactly once for every instance. In the case of BB-Rectangle, calling super-init performs Rectangle's initialization for the instance. BB-Rectangle provides two arguments to super-init because the Rectangle class consumes two initialization arguments.