Designing Grace: Can an Introductory Programming Language Support the Teaching of Software Engineering?

James Noble
Victoria University of Wellington, NZ
kjx@ecs.vuw.ac.nz

Michael Homer
Victoria University of Wellington, NZ
mwh@ecs.vuw.ac.nz

Kim B. Bruce
Pomona College, CA
kim@cs.pomona.edu

Andrew P. Black
Portland State University
black@cs.pdx.edu

Abstract

Many programming language constructs that support software engineering in the large — explicit variable declarations, explicit external dependencies, static types, information hiding, invariants — provide little benefit to the small programs written by novice programmers, where every extra syntactic token has to be explained and understood before novices can succeed in running even the simplest program. We are designing Grace, a new educational object-oriented language that we hope will prove useful for teaching both programming and software engineering. This paper describes some of the tradeoffs between teaching programming and teaching software engineering that we faced while designing Grace, and our attempts to address those tradeoffs.

1. Introduction

Programming language design involves a series of tradeoffs [8]. For example, we can make programs easier to compile by making them harder to write, for example, by requiring all constants to be declared together at the top of the program, and forbidding forward references [12]. We can make programs run faster by making them harder to write, for example, by requiring the programmer to choose between fast, fixed-sized integers or slow, arbitrary precision rationals. We can make programs less likely to fail at runtime, for example, by requiring programmers to catch all possible exceptions thrown by subroutines, or by imposing a strict static type discipline.

Tradeoffs of this kind are particularly problematic when the language being designed is intended to be used for teaching novices. Programmers certainly have strong opinions about the "best" first programming language [5]. The key tradeoff is between helping the student to get something working — often, *anything at all* working — and helping them to learn the *right way* to get something working. This tradeoff is especially acute when following a "software-engineering first" curriculum [15, 19].

Programming is to software engineering what carpentry is to building construction: a basic skill that is difficult to master, but that is not by any means the whole of professional practice. Many civil engineering practices — drawing blueprints, conducting quantity surveys, estimating costs, and complying with building codes — are crucial when constructing significant buildings or roadways. But these large-scale practices are unnecessary irrelevant distractions when building a treehouse in the backyard. In a similar way, many of the programming practices needed to support,

978-1-4673-5140-9/13 © 2013 IEEE CSEE&T 2013, San Francisco, CA, USA

and thus to teach, software engineering, are of most value when programming-in-the-large. We have in mind practices ranging from explicit variable declarations, explicit specification of external dependencies, compulsory static type specifications, information hiding, and formal behavioral specifications. Novice programmers don't typically start by programming-in-the-large: they begin by writing small programs. Even by the end of the first or second course on programming, many of the programming disciplines that promote good software engineering practice still make it harder for the students to get their small programs to run.

We are engaged in the design of a new object-oriented programming language named Grace (see http://www.gracelang.org). Grace aims to be suitable for teaching introductory programming courses, and to look familiar to instructors who know other object-oriented languages, while giving instructors and text-book authors the freedom to choose their own teaching sequences [2, 3]. One of our goals for Grace is to support the teaching of software engineering, at least during the first two years of professional formation, and then to act as a bridge to the languages that software engineers will use in practice [4]. At this stage in the project, we have a self-hosted Grace compiler that compiles to C and JavaScript [9], and we are experimenting with IDE support and developing teaching materials. We hope to begin small-scale teaching using Grace in the 2013–4 academic year.

The contribution of this paper is our reflection on some of the tradeoffs that we have faced during the design of Grace, in the spirit of Design Research [6]. Section 2 gives a brief overview of the design of Grace, reported in more depth elsewhere [2, 3, 11]. The following sections then present and discuss examples of particular language design tradeoffs in supporting software engineering.

2. Grace in a Nutshell

Grace is an imperative object-oriented language with block structure, single dispatch, and many familiar features [2, 3]. A single object is created by executing a particular kind of Grace expression called an object constructor:

```
object {
    def name is public, readable = "Joe the Box"
    var subBoxes := aList.empty
    method topLeft { 100@200 }              // @ is an infix operator
    method bottomRight { 200@400 }          // that creates a Point
    method width { bottomRight.x − topLeft.x }
    method height { bottomRight.y − topLeft.y }
    method addComponent(w) { subBoxes.push(w) }
    print "Joe the Box lives!"
}
```

The object created by executing this expression has methods topLeft, bottomRight, width, height, and addComponent. It also has a method name that acts as an accessor for the constant field name; this method is generated automatically because of the annotation **is** public, readable. The variable field subBoxes could have been annotated as readable and writable, but since it was not, it is private to the object. Creating this object has the side effect of executing the code inside the object constructor, which in this case prints `"Joe the Box lives!"`.

Grace's class construct defines an object with a single method that contains an object constructor; this method thus plays the role of a factory method:

```
class aBox.named(n:String)origin(tl:Point)diagonal(d:Point) {
    def name is public, readable = n
    var subBoxes := aList.empty
    method topLeft { tl }
    method bottomRight { tl + d }
    method width { bottomRight.x − topLeft.x }
    method height { bottomRight.y − topLeft.y }
    method addComponent(w) { subBoxes.push(w) }
    print "{name} the Box lives!"
}
var joeTheBox := aBox.named("Joe") origin(100@200) diagonal(100@200)
```

The class is called aBox, and its factory method is called named()origin()diagonal(): Grace allows method names to have multiple parts, like Smalltalk and Objective-C but without the colons. The declarations and statements between the braces (lines 2 to 9) describe the object created by this factory method; note how some of the methods capture state from the parameters of the factory method. Two other features of Grace are shown incidentally: method arguments that are already delimited, such as strings and numbers, need not be enclosed in parentheses, and strings can contain Grace code enclosed by braces. The value of such a string is computed by evaluating the code within the braces, requesting the resulting object to convert itself to a string, and interpolating that string in the place of the brace expression.

The last line illustrates a use of this class. Executing the expression on the right of the assignment will print "Joe the Box lives" and create a new object. The assignment will bind the variable joeTheBox to the new object.

Classes are completely separate from types: the class aBox is not a type and does not implicitly declare a type. The programmer may specify types if desired:

```
type Box = {
    name −> String
    topLeft −> Point
    bottomRight −> Point
    width −> Number
    height −> Number
    addComponent(w:Widget) −> Done
}
var joeTheBox:Box := aBox.named("Joe") origin(100@200) diagonal(100@200)
```

The type Done is like “unit”: it indicates that a method does not return a useful result. Grace types are structural: an object has a type if it responds to all of the methods in that type, with the correct argument and result types. It is not necessary for the object to have been “branded” with that type when it was created. Along with methods, types may be included as components of objects.

Variable and constant bindings are distinguished by keyword: **var** declares a name with a variable binding, which can be changed using the := operator, whereas **def** declares a name with a constant binding, initialised using =, as shown here:

```
var currentWord := "hello"
def world = "world"
...
currentWord := "new"
```

The keywords **var** and **def** are used to declare both local variables and fields inside objects. Visibility annotations allow the programmer to control access to methods from outside an object by marking them as public or confidential; the latter means accessible only to the object itself and objects that inherit from it. As we saw in the initial examples, annotations can also be used to

create reader and writer methods on fields. Method requests without an explicit receiver are resolved either as requests on **self**, or as requests on **outer**, the object in the surrounding lexical scope, on **outer**.**outer**, *etc*. If a receiverless request is ambiguous, the programmer must resolve the ambiguity explicitly.

Grace includes a special syntax for creating anonymous first-class functions. We call them blocks; other languages call them lambda expressions. Blocks are written between braces, and may be thought of as containing code for deferred execution. A block may have arguments, which are separated from the code by −>, so the successor function is {x −> 1+x}. A block can refer to variables bound in the surrounding lexical scope, and returns the value of the last-evaluated expression in its body.

The presence of blocks means that Grace's control structures can be defined as in libraries as methods. The familiar control structures, such as if...then...else and while...do, are defined in the standard prelude, but an instructor or library designer may replace or add to them. Grace's syntax is designed so that control structures look familiar to users of other languages:

```
if (x > 0) then { x } else { −x }

for (node.children) do { child −>
      process(child)
}
```

The use of braces and parentheses is not arbitrary: parenthesised expressions will always be evaluated exactly once, whereas expressions in braces are blocks, and may be evaluated zero, one, or many times.

Grace code at the top-level (of a file, or of the read-eval-print loop) is treated as if it were enclosed in an implicit object constructor. This allows methods and types to be defined at the top level; moreover, any code written at the top level will be executed immediately. As a consequence, Grace programs can be written in "script" form, without any object or class definitions at all, so `print "Hello, world"` is a compete Grace program.

Finally, Grace's module system provides support for dialects — supersets or subsets of the standard Grace language [10]. Naturally, dialects can extend the language by making extra definitions available: by defining new methods, the designer of a dialect can provide what look (to the programmer) like new statements and operators. Perhaps more surprisingly, dialects can also *restrict* the language: by providing extended static checkers, they can define and detect new classes of errors, and can change the way that errors are reported. Dialects can be used to define language subsets to aid novice programmers, as well as domain-specific languages customized for a particular purpose.

The idea of using families of language subsets to promote teaching and learning was introduced by DrScheme [20], now DrRacket. Racket uses the name "language levels": we use the term "dialect" rather than "language level" because we imagine a lattice of languages rather than a sequence. Unlike DrRacket's languages, which are built using macros, Grace's dialects are built as libraries that provide new definitions and static checkers; this means that Grace dialects cannot introduce new syntactic forms, or change the underlying language semantics. Dialects are one of the main techniques we use to reconcile opposing requirements in language design; this use of dialects is described in more detail in Section 8.

3. Syntactic Consistency *vs.* Semantic Consistency

Consistency is an important design principle, and applies as much to programming language design as to other kinds of design [13, 21]. Because students learning Grace will presumably transition to "industrial strength" programming languages, consistency is important not only within Grace itself, but also with respect to other programming languages. Thus, we chose to delimit Grace blocks by curly braces because this makes Grace superficially similar to the C family of languages, even though the semantics of Grace blocks are basically those of Smalltalk blocks — which are delimited by square brackets.

Consistency as a principle operates at multiple levels of language design. The choice of block delimiter is an instance of *syntactic consistency*, but we have also tried to consider *semantic consistency*: any particular piece of syntax should have a single semantics, as much as possible. Staying with delimiters, Grace's if...then and for...do statements take two arguments. Like languages in the C family, the first argument is in parentheses "()" and the second in braces "{}". This syntax happens to match the statement's semantics: the first argument, the test in the if statement, will be evaluated exactly once before the statement proper is invoked: the second argument is a block that may be executed zero, one or several times.

```
if (x > 0) then { print "x is bigger than zero" }
```

Grace's while...do statement does not follow this pattern: it takes two arguments, both of which must be delimited by curly braces. This is because both arguments are blocks, which may be executed repeatedly.

```
while {x > 0} do { print "x is bigger than zero for now"; x := x − 1 }
```

This design is syntactically inconsistent not only with Grace's other control flow statements, but also with the C language family generally. Here, Grace sacrifices syntactic consistency for semantic consistency: arguments that are passed by value, which are evaluated *before* requesting a method, are delimited by parentheses, whereas arguments passed by name, which may be evaluated (or not) under the control of the requested method, are consistently delimited by braces. We accept that this choice may make Grace marginally harder to learn, as students will have to learn the different patterns of braces and parentheses, and harder to transfer from Grace to C-like languages. We hope, however, that the overall simplification of the language, and particularly of its conceptual model of argument passing, is worth the tradeoff.

4. Static *vs*. Dynamic Types

In Grace, declarations of local variables, methods and fields are not required to specify types. Grace is gradually typed: omitted types of constants may be inferred, and omitted type of variables, arguments and results are treated as the predefined type Dynamic. Thus, in the declaration of class aBox in Section 2, methods width and height are treated as returning Dynamic, which conforms to any static type, including Number. All requests on Dynamic expressions are dynamically checked, as in C# [1]. In this way, Grace supports both statically and dynamically-typed code; indeed, programmers can choose at the level of an individual declaration.

Within dynamically-typed code, types need not be mentioned at all, and so all discussion of the concept of type can be delayed until late in the teaching sequence. When instructors do introduce types, they may do so in the language that students are already using, as opposed to, for example, starting teaching in Python and then being forced to transition to Java just so that types can be discussed.

To help an instructor ensure that students do indeed move to using static type specifications, a Grace dialect can be defined to require that all student programs written in that dialect have type specifications associated with all declarations; a different dialect might prohibit all type declarations [10]. Recall that Grace dialects cannot change the basic language semantics, or the interpretation of core language constructs such as declarations. The full Grace language is itself gradually typed, and the semantics of a program with no type specifications, partial specifications, or full specifications is always clearly defined. The checkers that are part of a dialect definition can be used to ensure that students use this flexibility in a disciplined way.

5. Explicit *vs.* Implicit Variable Declarations

Several contemporary scripting languages — even some, such as Python and JavaScript, that are promoted for use in introductory programming courses — allow the *implicit* declaration of variables. In these languages, as in Fortran or Basic, it is not necessary to declare a variable; the same syntactic form can be used to *assign* a new value to an existing variable, and to *declare* and initialize a new variable. This means that programs can be simpler. Unfortunately, it also means that a statement such as

```
countr := counter + delta
```

would declare a new variable countr rather than updating the existing variable counter.

In contrast, Algol, Pascal, and the C-family of languages require that the programmer declare variables explicitly, as does Grace. In these languages, declarations also specify the type of the variable, which may not then be changed by a subsequent assignment. Requiring explicit declarations guards against programmers accidentally creating a new variable (for example, because of a spelling mistake) when they intended to update an existing variable. Here is another case where simplicity — having assignment statements also create variables — may make programs shorter, and presumably easier to write; perhaps for this reason, most scripting languages have implicit variable declarations. Yet contemporary software engineering practice prefers the more complex, more verbose and less flexible solution with explicit variable declarations. Like most languages since Pascal, Grace allows variables to be initialized at the point of declaration; constants must be so initialized, since they cannot be assigned.

Grace also distinguishes between constants and variables, and requires explicit declarations for both. Constants are initialized using the = symbol; while variables are initialized and assigned using the := symbol.

```
def delta = 3        // declare and initialise a fresh constant
var count := 0       // declare and initialise a fresh variable
...
count := count + delta       // assign to an existing variable
```

While this distinction may make the language slightly more complex, we consider the clear conceptual separation worth the additional complexity. Also note that, in addition to reserving the symbol = for constant (and type) declarations, and := for variable initialization and assignment, we write the equality operator as ==. Again, we are willing to increase the syntactic overhead very slightly in order to gain the semantic consistency of using separate symbols for separate concepts.

6. Information Hiding *vs.* Direct Access

Information hiding has been recognized as an important software engineering practice for the last forty years. Object-oriented languages typically support information hiding by controlling access

to an objects' fields and methods. Typically, fields and methods may be private — accessible only from within that object — or more accessible, perhaps from other objects of the same class, or from objects defined within the same package, or defined in a particular program component. Another of Grace's design principles is to encourage software engineering best practice: well-engineered programs should be as easy — or easier — to write than badly-engineered programs. In this case, best practice would be to make an object's variables, fields, and methods private by default. Unfortunately, this default would make simple programs more difficult for novice programmers to write. Consider the following example, which does nothing more than create an object and then print it:

```
def joe = object {
    var forename := "Joe"
    var surname := "Bloggs"
    var id := 234567
    method asString {"Person: {forename} {surname} id:{id}"}
}

print "joe is {joe}." // error here
```

This example is erroneous because the "print" method requests joe.asString to compute the value of the interpolated string. This request comes from outside the object; because methods are private by default, the request will be denied.

The programmer can easily change the code to make the asString method accessible from outside by annotating it as "public". Annotations can also be applied to definitions, where they can be used to create reader methods, and to variable declarations, where they can be used to create reader and writer methods.

```
def jose = object {
    var forename is public, readable := "Jose"
    var surname is public, readable := "Bloggs"
    var id := 234567
    method asString is public {"Person: {forename} {surname} id:{id}"}
}

print "jose' is {jose}." // now works
```

Annotations clutter the code, and mean that novices must learn one more thing before they can successfully use an object. More significantly, requiring public annotations links the concept of defining a method with the concepts of information hiding and and access control: because some methods (or fields) have to be declared public before objects can be used at all, the concept of information hiding must be learned before, or concurrently with, the concept of objects. Learning concepts that are coupled in this way may be harder than learning independent concepts [18]. While making methods and fields private by default has better software engineering properties, it may also make programming harder to learn.

Unlike some of the other tradeoffs described in this paper, this one seems harder to finesse via libraries or dialects. Objects, methods, variables and constants are the primary constructs in Grace, and their semantics, including their accessibility, must be defined in the core of the language. The current compromise is to make methods public by default, and to allow private annotations to *restrict* access. In contrast, variables and constants would be private by default, and annotations would provide a succinct way of making them accessible. This would permit the unannotated joe example above to run. While obviously inconsistent, the rationale behind this compromise is that programmers' first objects are generally simple; the only reason for providing those objects

with methods is to allow clients to request those methods. The need for private methods does not become apparent until students start to write complex methods that need to be decomposed into a series of requests of helper methods; at that point, the private annotation can be introduced. In contrast, decisions about the concrete representation of even simple objects should be hidden from their clients, so fields should be private by default.

7. Formal *vs.* Informal Reasoning About Code

One of the key advantages of languages like Eiffel for teaching software engineering [14, 15] is that they contain language constructs to support assertions, invariants, and variants. Like many languages, Grace offers an assert keyword that takes a block and raises an error if the assertion is invalid: unlike many languages, in Grace this defined in the standard library as a method that accepts a block as an argument:

```
assert {(letters.size > 0) && (letters.size < 20)}      // assertion

method assert (block) {                                 // implementation of assert
    if ( ! block.apply) then {error "Assertion Failed"}
}
```

Pre- and post-conditions and object invariants can also be provided in libraries by similar techniques. Grace is not alone here, as both C# and Scala offer similar libraries — although their implementation is more complex than the Grace version [7, 17].

Eiffel includes more support for formally specifying program properties than most other languages — certainly more than C# or Scala contracts, in that its loop statement allows programmers to specify loop variants and invariants that are checked when the program runs. Using methods and blocks, similar facilities can be implemented in Grace, as illustrated by this example:

```
method gcd(m, n)
// Euclid's Algorithm for greatest common divisor
{
    assert {(m >= 0) & (n >= 0) & ((m != 0) | (n != 0))}
    var a := max(m,n)
    var b := min(m,n)
    while {b != 0}
        invariant { a >= b }
        do {
            def remainder = a % b
            a := b
            b := remainder }
        variant {b}
    return a
}
```

Furthermore, students can even inspect the code that implements these language features (17 lines including a 4 line method header [16]). This is another illustration of how a small set of carefully-chosen mechanisms can support a range of teaching approaches, without increasing the complexity of the core language.

8. Dialects *vs.* a Single, Uniform Language

Grace's dialects are one of the key techniques that we have used to resolve tradeoffs in designing for teaching programming versus teaching software engineering. As we have explained, dialects

are specialized languages that can provide extra definitions, but can also restrict the availability of certain language features. As with Racket, Grace is less a single language than a "product line" of languages tailored to particular educational contexts. Dialects allow students to be provided with a language suited to their own stage of learning. Thus, a beginning student dialect might make certain language features unavailable, and might thus be able to improve the quality of error messages; this can be very helpful to students. Moreover, such a restricted dialect can be provided without limiting the entire language to the same level. Dialects also allow specialized languages for particular tasks or problem domains to appear first-class.

The very existence of dialects creates a new problem, however: there may be many incompatible dialects, and thus programmers need to specify which dialect they are using. Should this specification be explicit in the code, or implicit in the programming environment? Making the choice of dialect explicit is arguably the right thing to do from a software engineering perspective, since it makes a dependency (on the dialect definition) explicit in the source. This would place an additional burden on novices, who, at least at the beginning, are unlikely know about dialects at all. Moving the selection of the dialect to a menu in the IDE makes things simpler for the novice — until she reads ahead in the textbook, and tries an example program (whiten in a more advanced dialect), only to find that it won't compile.

We have not found a simple solution to this problem, but ameliorate it in two ways. Because dialects can only define new methods and restrict the use of other features, a program in any dialect is syntactically-valid Grace, and has a clear meaning, even if it does request methods that appear to be undefined. While we require that the dialect in use be explicitly declared in the source code, we keep dialect specifications very brief, to minimize the cognitive load for new programmers. Moreover, an IDE for novice students can enforce the use of a particular dialect, at the discretion of the instructor.

These tradeoffs have led us to a dialect system that works well in many cases, but could benefit from more power in others. It works well for a dialect to enforce the use of loop invariants, since all that dialect need do is define new control structure methods with invariant checks. A dialect to enforce static typing also works well. However, a dialect cannot provide *additional* type inference, beyond that provided by the base language. By using libraries to define dialects, rather than macros, we have been able to preserve a uniform language while retaining the ability to extend it naturally.

The key features of Grace's design that enable this specialization are not the details of the dialect mechanism itself [10], but some more basic design choices that underpin dialects: defining every control structure (syntactically and semantically) as a method request, using blocks to pass code into those methods, and allowing nesting with lexical scope in all contexts. The power of these choices shows the value of simple mechanisms used consistently.

9. Conclusion

Grace is a new object-oriented programming language that we are designing for use in education — both for learning programming and for learning software engineering. We have described some of the design decisions reflected in the current version of Grace, focussing on where those decisions addressed tradeoffs between making programming easy and supporting a disciplined approach to software engineering. We have argued that many — but not all — of these tradeoffs can be resolved within a range of dialects that allow instructors to allow or disable particular language features.

We plan to begin experimental use of Grace for teaching in the 2013–14 academic year, and hope to conduct systematic evaluations of Grace in that context. No doubt that experience will continue to influence our design choices as the language continues to evolve.

Acknowledgment

This work is supported by the Royal Society of New Zealand Marsden Fund, and the School of Engineering and Computer Science and Victoria University of Wellington, New Zealand.

References

[1] G. M. Bierman, E. Meijer, and M. Torgersen. Adding dynamic types to C#. In *European Conference on Object-Oriented Programming (ECOOP)*, 2010.

[2] A. P. Black, K. B. Bruce, M. Homer, and J. Noble. Grace: the absence of (inessential) difficulty. In *Onward!*, 2012.

[3] A. P. Black, K. B. Bruce, M. Homer, J. Noble, A. Ruskin, and R. Yannow. Seeking Grace: A new object-oriented language for novices. In *ACM Conference on Computer Science Education (SIGCSE)*, 2013.

[4] A. P. Black, K. B. Bruce, and J. Noble. Panel: designing the next educational programming language. In *SPLASH/OOPSLA Companion*, 2010.

[5] cplusplus.com. What was your first Programming language? http://www.cplusplus.com/forum/lounge/48273/, Aug. 2011.

[6] N. Cross. Creative cognition in design: Processes of exceptional designers. In T. Hewett and T. Kavanagh, editors, *Creativity and Cognition*. ACM Press, 2002.

[7] M. Fähndrich, M. Barnett, and F. Logozzo. Embedded contract languages. In *ACM Symposium on Applied Computing (SAC)*, 2010.

[8] C. Hoare. Hints on programming language design. Technical Report AIM-224, Stanford Artificial Intelligence Laboratory, 1973.

[9] M. Homer. Minigrace to JavaScript compiler. http://ecs.vuw.ac.nz/~mwh/minigrace/js/, 2011.

[10] M. Homer, J. Noble, K. B. Bruce, and A. P. Black. Modules and dialects as objects in grace. Technical Report ECSTR13-02, School of Engineering and Computer Science, Victoria University of Wellington, Mar. 2013. `http://ecs.victoria.ac.nz/Main/TechnicalReportSeries`.

[11] M. Homer, J. Noble, K. B. Bruce, A. P. Black, and D. J. Pearce. Patterns as objects in Grace. In *ACM Dynamic Language Symposium (DLS)*, 2012.

[12] K. Jensen and N. Wirth. *Pascal User Manual and Report*. Springer, 1975.

[13] B. J. MacLennan. *Principles of Programming Languages: Design, Evaluation, and Implementation*. OUP, 1995.

[14] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[15] B. Meyer. *Touch of Class: Learning to Program Well with Object and Contracts*. Springer-Verlag, 2009.

[16] J. Noble. Laissez-parlent l'Eiffel! (Grace language weblog entry). http://gracelang.org/applications/2013/01/28/laissez-parlent-leiffel, Jan. 2013.

[17] M. Odersky. Contracts for Scala. In *Runtime Verification (RV)*, 2010.

[18] A. Robbins. Learning edge momentum. *Computer Science Education*, 20(1):37–71, Mar. 2010.

[19] The Joint Task Force on Computing Curricula. SE2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering. http://sites.computer.org/ccse, Summer 2004.

[20] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. In *ACM Conference on Programming Languages Design and Implementation (PLDI)*, 2011.

[21] R. Williams. *The Non-Designer's Design Book*. Peachpit Press, 1994.