Object-Oriented Implementation Approaches of Pure Object-Oriented Languages:

A Comparison among Smalltalk, Eiffel, Ruby and Io

Christopher Bowen, Kevin Desmond, Jesse Kurtz, Jack Myers

Abstract. Smalltalk, Eiffel, Ruby and Io are all prime examples of pure object-oriented languages. Their implementation of such object-oriented features such as inheritance, encapsulation, polymorphism and abstraction differ, however. To assess the variations in these languages, the authors developed a rudimentary class registration system in Smalltalk, Eiffel, Ruby and Io. The mechanics and syntax of these features will be illustrated through a series of examples. When a feature does not fully exist, but a work-around can be crafted, this will also be discussed.

1. Introduction

Smalltalk, Eiffel, Ruby and Io are all prime examples of object-oriented (OO) languages that were incarnated in subsequent decades as they were first released in 1972, 1986, 1995 and 2002 respectively.

The creation of Smalltalk is credited to Alan Kay and his team at Xerox PARC from 1971 to 1975. The language was created based on two main ideas. The first idea introduced the concepts of classes and messages from Simula. The second principle was that the language would have no fixed syntax. Originally, each class controlled its own behavior. This quickly became a nuisance, and by 1976 a completely new version of Smalltalk had been implemented. The idea of inheritance had been added to Smalltalk along with a fixed syntax which made for faster, simpler, and more readable programs (Hunt 45).

Eiffel was developed by Bertrand Meyer and his company, Eiffel Software (previously known as Interactive Software Engineering Inc.) in 1985. The language itself was started on September 14th of that year and introduced to the public in October of 1986. Eiffel was named after Gustave Eiffel, the engineer who designed the Eiffel Tower, because the developers wanted to maximize complexity, transparency, and stability – much like the tower itself. Its most important contribution to software engineering is *design by contract*, in which preconditions, postconditions, assertions, and class invariants are used to help ensure correctness without losing efficiency ("The Eiffel Programming Language).

Ruby was developed by Yukihiro Matsumoto who was unsatisfied with the difficulty and complexity of other languages (Flanagan and Motsumoto 2). Matsumoto stated that he wanted a scripting language that was "more powerful than Perl, and more object-oriented than Python" (Stewart).

Io is a small, compact language without widespread and accessible scholarly commentary. The Guide section of Io's main website, IOlanguage.com, contains the most extensive information on the language. Another good reference site is http://io-fans.jottit.com/ which consists of multiple links to Io resources.

Io does not enjoy a large audience. Those who encounter Io tend to get excited, get involved, and then move on with their computer programming language pursuits. Yet renewed interest in the Io language has been sparked by Bruce Tate's 2010 book entitled “7 Languages in 7 Weeks” (Buday).

1.1 Purities and impurities

The exact definition of what makes an object-oriented programming language "pure" varies slightly depending on the perspective of the computer scientist. Meyer notes, "'Object-oriented' is not a boolean condition: environment A, although not 100% O-O, may be 'more' O-O than environment B" ("Object-Oriented" 22). C++ creator Bjarne Stroustrup dismisses the concept of purity as an

inference that any language that implements non-object-oriented features in addition to abstraction, inheritance and run-time polymorphism is less robust than one that adheres strictly to the OO paradigm. Stroustrup remarks, "not everything good is object-oriented, and not everything object-oriented is good" (Stroustrup).

Stoustrup's perspective may likely be influenced by the fact that C++ allows for primitive object types like *int* and *float* due to a design principle that mandated backward compatibility with C. Java, also not a pure OO language, utilizes primitives as well. It has been noted that mixing OO and non-OO features into a programming language can result in faulty code, as the mechanics of the language operators will vary based on variable type. The Java example below indicates the variance.

```
double d1 = 5.5;
double d2 = 5.5;
Double D1 = new Double(5.5);
Double D2 = new Double(5.5);

System.out.println(d1 == d2);
System.out.println(D1 == D2);
```

The == operator can either return true if the variables compared contain the *same value* (for primitives d1 and d2), or true only if the *identity* of the object-based variables is the same. Therefore the expression D1 == D2 would return false.

Such impurities can lead to surprising results that baffle novice programmers. Consider how Java handles String objects, as shown below.

```
String str1 = new String("hello");
String str2 = new String("hello");
String str3 = "hello";
String str4 = "hello";

System.out.println(str1 == str2);
System.out.println(str3 == str4);
```

One would expect Java's equal-to relational operator == to perform an identity test on String objects as it functioned previously with Double objects and return false for both println statements. Yet Java returns true for the second statement, as its compiler generates code to have str3 and str4 reference the exact same string instance.

Despite Stroustrup's contention, a mixed language does allow for a number of semantic ambiguities. Admittedly, programmers can learn the behavior of their language and compensate accordingly, but why should they have to? It is logical to expect str3 be "equal to" str 4. Smalltalk, for instance, handles object equality elegantly by implementing a similar technique. Hidden in Smalltalk's implementation, a class determines if an instance with the same value already exists (Alpert). If so, Smalltalk returns that object instance such that a more logical interpretation of equality can be maintained.

Io claims to have primitives. On closer inspection, though, it is revealed that all Io primitives inherit from the Object prototype and have methods which are mutable (iolanguage.com). This implementation is in stark contrast to the true primitives of C++ and Java.

1.2 Comparative Methodology

While all four languages covered in this article are considered to be pure object-oriented languages, they differ greatly in how the object-oriented features are implemented.

To illustrate the differences between the languages, a rudimentary class registration application was implemented in each. The actors will be Students who will register for sections, Instructors who will be assigned to teach sections, Teaching Assistants who may act as both a student and a teacher, and Registrars who manage the system.

For sake of simplicity, only Registrars will be able to enroll or drop Students (or Teaching Assistants acting as Students) from Sections. Registrars will also be able to upload, add and delete Courses, Instructors, Students and Teaching Assistants. A use case diagram depicting these interactions is show in Figure 1.

A class structure was designed both to model a real-world situation and provide an opportunity to test several object-oriented features of the languages. Properties and methods of the classes are listed in Figure 2.

As object-oriented languages, Smalltalk, Eiffel, Ruby and Io each include inheritance, encapsulation, polymorphism and abstraction. As pure OO languages, they each adhere to the following rules:

- all pre-defined types are objects (i.e., no primitive data types exist);
- all user-defined types are objects (i.e., programmers cannot create a user-defined primitive type);
- all operations are messages to objects.

Inheritance, encapsulation, polymorphism and abstraction allow for a more tangible and interesting comparison than the previous three requirements for purity. Technical requirements were developed to force an implementation of a particular OO technique. In the sections to follow, selected aspects of each of these features will be compared and contrasted across the four languages.

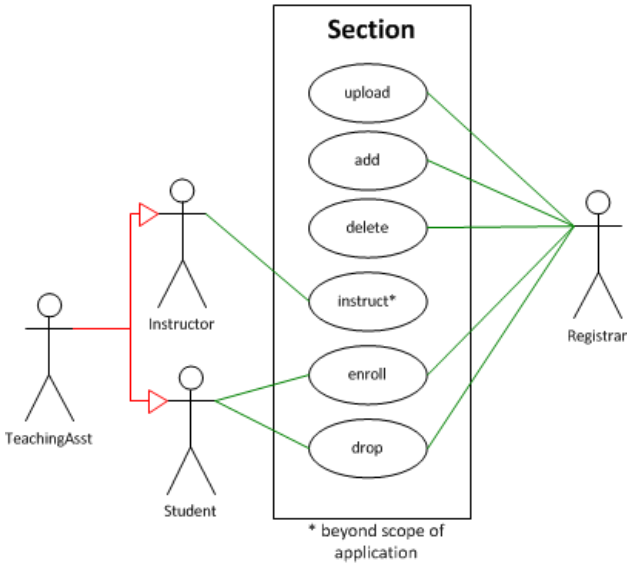

Fig. 1 Use case diagram for registration application

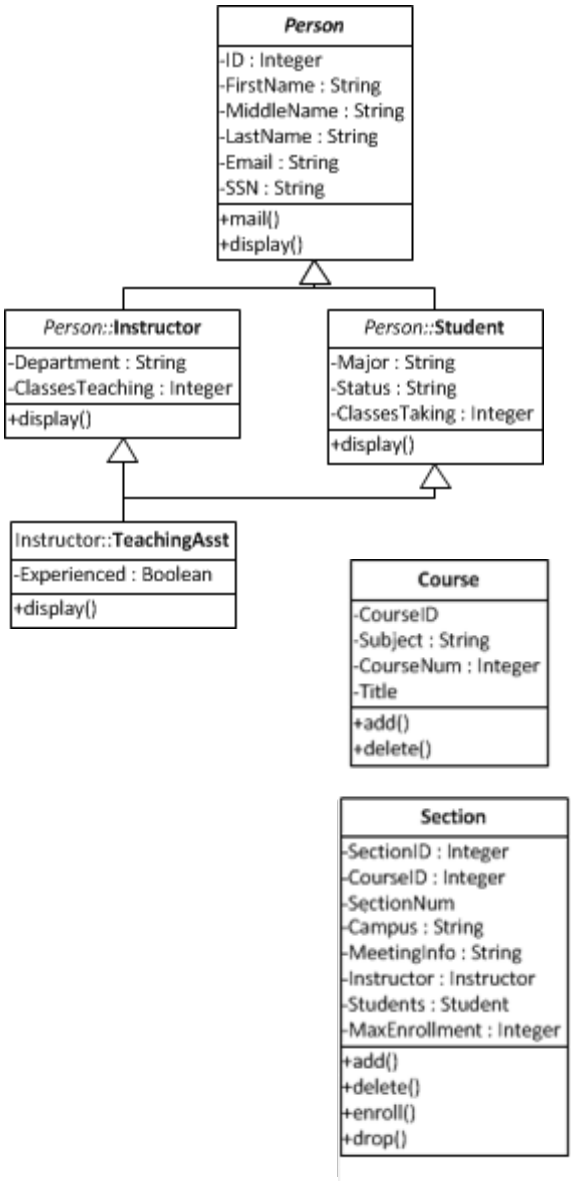

Fig. 2 Classes, properties and methods in application

2. Inheritance

2.1 Multiple Inheritance

In the specifications for the sample application, a teaching assistant needed to be a subclass of both Student and Instructor.

An object-oriented language that allows multiple inheritance must semantically deal with the potential clashes that could arise from inheritance of similarly named properties and methods from multiple parents.

Eiffel does allow a class to inherit from any number of different classes as it needs. For example, a class TEACHING_ASSISTANT is able to inherit from both INSTRUCTOR and STUDENT classes.

```
class
   TEACHING_ASSISTANT
inherit
   INSTRUCTOR
   STUDENT
```

The TEACHING_ASSISTANT class uses the inherit declaration clause to define what its parent classes are. There are a few built-in feature adaptations that handle any possible conflicts between parent features. These adaptations will be described in subsequent sections.

There is another implementation challenge with multiple inheritance. A teaching assistant inherits from both INSTRUCTOR and STUDENT. These two classes, however, also each inherit from PERSON. This condition is known as repeated inheritance which raises the question: what do the features of repeated ancestors do for the repeated descendant? There are two possibilities for this:

- sharing (The repeatedly inherited feature yields just one feature.);
- duplication (The repeatedly inherited feature yields two features) ("The Eiffel Programming Language").

For example, suppose instructors and students each have different types of computer accounts (INSTRUCTOR. computer_account and STUDENT.computer_account, respectively) and TEACHING_ASSISTANT inherits both of them. If they are renamed by TEACHING_ASSISTANT (to faculty_account and student_account), there is no conflict. However, if not renamed, the call to myTeachingAsst. computer_account is then ambiguous. Using *select* will apply to call to the selected feature.

```
class
   TEACHING_ASSISTANT
inherit
   INSTRUCTOR
      rename
```

```
        computer_account as faculty account
    select
        faculty_account
    end
  STUDENT
    rename
        computer_account as student_account
    end
```

In this example, myTeachingAsst.computer_account would have only one computer_account, i.e., faculty_account inherited from INSTRUCTOR. The select phrase is only needed in case of replication.

Io, being a compact language, is best understood in context of what constructs are missing. There are no globals; there are only locals. There are no classes. Thus, there is no distinction between instances and subclasses. Io is a *prototype based* language, where prototypes serve as both instances and classes (Ansaldi). However, there is a round-about way to create a subclass.

The principle of unification is a central tenet of Io. Many elements of computer linguistics are unified into this language structure which consists solely of objects. Prototypes (or "protos") are an extension of this objectified unification concept. There is a list of protos associated with each object. Gaining access to the various slots within the list of protos identifies the values unique to the given object.

Despite being classless, multiple inheritance is a feature of Io. Iolanguage.com states, "You can add any number of protos to an object's protos list. When responding to a message, the lookup mechanism does a depth first search of the proto chain" (iolanguage.com)

Unlike Io, Ruby has classes, yet it does not permit a class to inherit from two different classes. Therefore a decision had to be made whether TeachingAsst would inherit from Student or Instructor. However, multiple inheritance can be simulated using a Ruby technique called *mixIn modules* (Thomas, Fowler and Hunt 383). A module Pupil (synonym for student) was created. This module contains a method to display enrolled classes as well as the instance variables that would have been placed in Student.

```
module Pupil

  def displayClasses(classAttendee)
    puts "Classes for {#classAttendee.firstname}
          {#classAttendee.lastname}:";
    # Logic to retrieve and display data
  end
end
```

The TeachingAsst class utilizes the include statement to denote that a teaching assistant is a Pupil, even though it is an official subclass of Instructor.

```
class TeachingAsst < Instructor
  include Pupil
```

The Student class contains a similar statement. In effect, this allows a TeachingAsst to be both a Pupil and an Instructor, as seen in the following code:

```
jack = Student.new("Jack","F.","Myers" …)
jesse = TeachingAsst.new("Jesse","O.","Kurtz" …)

puts " jack is a student.
         #{jack.is_a? Student}"
puts " jack is a pupil.
         #{jack.is_a? Pupil}"
puts " jack is an instructor.
         #{jack.is_a? Instructor}"
puts " jack is a TA.
         #{jack.is_a? TeachingAsst}\n\n"
puts " jesse is a student.
         #{jesse.is_a? Student}"
puts " jesse is a pupil.
         #{jesse.is_a? Pupil}"
puts " jesse is an instructor.
         #{jesse.is_a? Instructor}"
puts " jesse is a TA.
         #{jesse.is_a? TeachingAsst}"
```

```
jack is a student.      true
jack is a pupil.        true
jack is an instructor.  false
jack is a TA.           false

jesse is a student.     false
jesse is a pupil.       true
jesse is an instructor. true
jesse is a TA.          true
```

Fig. 3 Simulation of multiple inheritance in Ruby

As seen in Figure 3, note that *jesse* is a bona fide Ruby Pupil, in addition to being a TeachingAsst and an Instructor, but there was no way in Ruby to let *jesse* inherit from more than one class.

However, in contrast to Eiffel, such "multiple inheritance" fails to address the situation in which both the superclass and the module contain the same property name, but their setters differ.

```
class Superclass

  def initialize(id)
    @id = id
  end

  def id=(i)
    @id = i
  end
```

```
module Superclass2

  def id=(i)
    @id = i * 100;
  end
```

```
class Subclass < Superclass
```

```
  include Superclass2;

  def initialize(id,empNum)
    super(id)
    @empNum = empNum
  end
```

In the Ruby simulation of multiple inheritance, the private initialization method of the superclass would have to be invoked when instantiating a subclass. However, the accessor method to set the property would be that of the module Superclass2, as the include statement for that module would override the superclass's setter.

Thus, a confusing situation would arise as in the code below, where line 21 results in "ID = 3", but line 23 results in "ID = 300".

```
20 test = Subclass.new(3,409)
21 puts "ID = #{test.id}"
22 test.id = 3
23 puts "ID = #{test.id}"
```

Like Ruby, Smalltalk also does not support multiple inheritance. However, the language does support a construct known as a *trait* which functions similarly to multiple inheritance. A trait is a set of methods that can be reused by any number of classes (Scharli, Ducasse, Nierstrasz and Black). For the class registration example there will be a trait TPupil that will be used by both the Student and TeachingAssistant classes:

```
Trait named: #TPupil
     uses: {}
     category: 'RegistrarSystem'
```

It is common convention in Smalltalk to begin trait names with a capital 'T' to differentiate them from classes. It is possible for a trait to use another trait, so the line "uses: {}" is included in the trait definition. In this example, however, no other traits are being used. The Student and TeachingAssistant classes will include a similar line of code in their class definitions:

```
Person subclass: #Student
     uses: TPupil
     instanceVariableNames: 'major status classesTaking'
     classVariableNames: ''
     poolDictionaries: ''
     category: 'RegistrarSystem'

Instructor subclass:  #TeachingAssistant
     uses: TPupil
     instanceVariableNames: 'experienced'
     classVariableNames: ''
     poolDictionaries: ''
     category: 'RegistrarSystem'
```

The TeachingAssistant class is also a subclass of Instructor. By implementing Instructor and using the TPupil trait, multiple inheritance is simulated in the project. In this way the traits of Smalltalk function similarly to the modules of Ruby.

When working with traits in Smalltalk, it is common to give aliases to trait methods within a class. This is to prevent conflicts between methods of the same name. For example, the Student class contains a getter method, denoted as #major, which returns the major of that particular instance of Student:

```
"Student class"
major
    ^major
```

This #major method will not ever be called directly. Instead, this method will be accessed through the trait TPupil. The trait will include a similar function, also named #major:

```
"TPupil trait"
major
    ^ self major
```

This method sends a message to #major of the Student class. A conflict arises here because both methods are given the same name. If this message is sent to a student, #major of the Student class will be executed. This is not the desired response since the use of traits is not actually demonstrated. The class definition for student can be changed to:

```
Person subclass: #Student
     uses: TPupil @ {#myMajor -> #major}
     instanceVariableNames: 'major status classesTaking'
     classVariableNames: ''
     poolDictionaries: ''
     category: 'RegistrarSystem'
```

Now, #major of TPupil can be referenced as #myMajor for any instance of Student. Presumably, from now on, only the message #myMajor will be sent to a Student, making use of the TPupil trait. This same approach can be used to deal with all the getters and setters in the TeachingAssistant and Student classes that pertain to TPupil. Each alias is separated by a period in the uses statement, for example:

```
uses: TPupil @ {#myMajor -> #major. #myStatus -> #status}
```

2.2 Interfaces

In Java, an interface is a group of related methods with empty bodies. A class that implements an interface must have method definitions for every method in the interface or the class cannot compile. The major advantages of interfaces are:

- the enforced rigorous application of an object model;
- creation of reusable sets of related methods that can be flexibly applied to multiple objects.

In the registration application, one requirement mandates the use of an interface to validate name properties which appear in the Person, Student, Instructor and TeachingAsst classes.

Since Ruby is a scripting language and compiled at run-time, there can be no enforcement that subclasses implement all interface methods. However, the principle of interface reuse is accomplished using mixIn modules in the same fashion as Ruby mimicked multiple class inheritance. Module NameValidator was included in the Person class and properly inherited by its subclasses.

Name values are compared to Ruby regular expressions `/^[\p{L}][\p{L}\-\'\.\s]+/` and `/\s{2,}/`. The first expression matches strings that begin with a UTF-8 letter followed by any number of UTF-8 letters, a hyphen, period or a space; the second checks for the invalid condition of multiple adjacent spaces.

Smalltalk does not explicitly support interfaces as Java or Objective-C; however, there is an interesting way to work around this and create explicit Interfaces using Smalltalk itself. Interface could be created as a super class, with all other interfaces being its subclasses. Smalltalk developers Benny Sadeh and Stephane Ducasse describe how static and dynamic interfaces can be built in Smalltalk (Sadeh and Ducasse). This type of language extension is evidence of Smalltalk's extensibility, in that everything in Smalltalk is an implementation feature. For further reading on this topic, see "Adding Dynamic Interfaces to Smalltalk."

Io does not have interfaces. Yet, despite this deficit, other dynamic means can be employed to separate implementation from behavior. Those means include the dynamic nature of adding methods in object slots, modifying methods in object slots, and invoking methods in object slots.

The Io language is prototype-based, as described in the Io Programming Guide:

> In Io, everything is an object (including the locals storage of a block and the namespace itself) and all actions are messages (including assignment). Objects are composed of a list of key/value pairs called slots, and an internal list of objects from which it inherits called protos. A slot's key is a symbol (a unique immutable sequence) and its value can be any type of object (Io Programming Guide).

It is the dynamic nature of the slot and method interaction which bears the 'spirit,' so to speak, of interfaces in the Io language.

Eiffel also does not explicitly have an "interface" declaration. Eiffel defines an interface as code which communicates to other programming languages, not from one source file to another. The functionality of a class that would be considered as an interface in other languages would be defined using feature renaming, and property visibility, discussed later in subsequent sections of this paper.

2.3 Feature Renaming

In a registrar system, an instructor would contain a field for the number of classes they are teaching, called ClassesTeaching. However, a teaching assistant is not actually teaching a class but rather assisting the instructor. In subclass TeachingAsst, the "ClassesTeaching" property might be more contextually correct if it were to be aliased "ClassesAssisting." This OO concept is known as feature renaming.

Knowing that in Ruby, access to class properties is achieved exclusively through methods, a property can be virtually renamed by creating method aliases for a getter and setter as shown below from class TeachingAsst.

```
# Create an alias for ClassesTeaching from superclass
alias :classesAssisting :classesTeaching
alias :classesAssisting= :classesTeaching=
```

Once established, either the alias methods or the original ones in Instructor can be invoked. Hence both myTeachingAsst.classesTeaching and myTeaching-Asst.classesAssisting will properly evaluate to the same value.

The Ruby implementation of feature renaming is not as complete as Eiffel's implementation, because Ruby retains the original feature name as well as the alias. Eiffel, on the other hand, has a more rigorous implementation of feature renaming as would be expected in a language that truly supports multiple inheritance.

As discussed previously, multiple inheritance can cause name clashes when two parents include a feature with the same name. This conflicts with the rule that no two features of a class may have the same name, or name overloading. Eiffel provides a way to avoid this issue through the *rename* subclause.

Every feature of an Eiffel class has a *final name*. For a feature introduced in the class itself, the final name is the name appearing in the declaration. This final name would

be the feature's name in the parent class for inherited features that are not renamed. This results from the Final Name Rule which states "Two different features of a class may not have the same final name" ("Invitation").

In Eiffel, it is important to understand the difference between *renaming* and *redefining*. Renaming alters the name of the feature but preserves its functionality. Redefinition keeps the name constant but changes the feature. When declaring a feature as renamed, it is important that it appears before all other subclauses such as redefine, export, undefined, and select.

It is not uncommon to inherit features from two different parents with the same name, provided they have the same signature. In this case, the features are joined, or merged into one feature. This is done by logically combining the preconditions with "or" operations and the postconditions with "and" operations. This does not violate the previously mentioned Final Name Rule since both are combined into a single feature. There may, however, be a time when one of the features should be used instead of the union of them. This is done using the *undefine* subclause:

```
class
   D
inherit
   A
      rename
         g as f
      undefine
         f
      end
   B
      undefine
         f
      end
   C
      -- C also has an effective feature f, which will
      -- serve as implementation for the result of the
      -- join.
feature
```

As seen in the above example, the three inherited classes all have an implementation of a feature named "f." By undefining the feature in both A and B, the implementation of "f" in C would take over.

Feature renaming for instance variables and methods of a class is not supported in Smalltalk. Trait methods, on the other hand, can be aliased as shown before. The concept is similar to the feature renaming used for some of the other languages to rename classesTeaching to classesAssisting for a TeachingAssistant.

Like SmallTalk and Ruby, renaming is only possible in Io through aliasing, an example of which is shown below.

```
MyObject someMethod := method(
        doSomething
)

//alias someMethod

MyObject someMethodWithoutSomethingNew := MyObject
getSlot("someMethod")
MyObject someMethodWithSomethingNew := method(
        doSomethingNew
        someMethodWithoutSomethingNew
)
```

Aliasing gives the Io language a limited ability to feature rename.

3. Encapsulation

3.1 Property Visibility

In Java, in addition to the traditional private properties with public getters and setters, public properties are allowed even though they violate the principle of encapsulation. A protected property is insulated to a controlled segment of an application. Imagine if in release one of an application, the property age in class Person was of float data type to store ages such as 9.5. In release two, when age was changed to an integer to avoid storing partial years, only the calculations in Person and its subclasses (including setters) would need to be rewritten. Other segments of the application would be protected from the change to the property.

In Io, everything is public, making property visibility a moot concept, as use of properties is very straight forward. This is evident when considering message sending. When sending a message to an object, the object responds if it contains a slot with the same name as the message. If there is no slot with the message name in that hash table, the search moves to the next prototype in the list. The search continues, until the slot in question is found. If the slot is not found, the forward slot is used, or an error is generated, if there is no forward slot.

Eiffel has quite a sophisticated syntactical construct which organizes feature declarations are into groups according to their visibility. This implementation allows a class to control where clients may call its features. Each group has a header that lists which classes may use the features within that section. If a header has no list, then any client may call the features. Similarly, if a list is declared but is empty or contains the NONE keyword, then all of the features are for internal use to the class only.

Inside a class itself, all of its own features are always visible. For example, the Instructor class would not need to know a student's social security number, but would benefit from having access to their email. In contrast, the Registrar class would need the student's SSN; therefore the

Student class could configure such access as shown below.

```
class
   STUDENT
inherit
   PERSON
feature
   email: STRING
feature {REGISTRAR}
   ssn: STRING
   do
      Result := ssn
   end
```

In this case, if an instructor were to attempt to call STUDENT.ssn, an error stating the feature of qualified call is not available to client class, but the Registrar would have no problem doing so.

A scenario could exist where an instructor would need to know the SSN of a teaching assistant. This would not be immediately possible, since Eiffel inherited this feature from the Student class. Using simple Eiffel inheritance, the feature could be re-declared; however, Eiffel has a more elegant option – utilizing the *export* phrase. The export phrase would be used in the inheriting class to expand the visibility of a property inherited from the parent. The code below demonstrates how the TEACHING_ASSISTANT class could expand SSN visibility to the INSTRUCTOR class.

```
class
   TEACHING_ASSISTANT
inherit
   INSTRUCTOR
   STUDENT
   rename
      ssn as tassn
   select
      tassn
   export
      {INSTRUCTOR} ssn
   end
```

Smalltalk does not have any explicit form of privacy. Conventionally, if a method is meant to be used privately, the programmer can include the word “private” in the comments. Smalltalk trusts its programmers to use these methods as if they were actually private.

In Ruby, all instance variables are fully encapsulated and are effectively private. (Flanagan and Matsumoto 232) Consider the program below:

```
1 class Demo
2   jack = Student.new("Jack","F.","Myers",
                "jack@gmail.com","111-22-3333",
                "Art","Graduate")
3   print jack.major + "\n"
4   jack.major = "Computer Science"
5   print jack.major
6 end
```

For those familiar with Java, it looks like the "property" major from instance jack is being directly accessed in line 3 and directly set in line 4. However, this is misleading. In fact, major is not a property at all, but rather a *method*. Statement 3 and Statement 5 are both method calls from class Student.

```
class Student

  def major
    @major
  end

  def major=(m)
    @major = m
  end
```

Line 3 invokes method *major*, which returns the value of the instance variable @major. (In Ruby, the returned value is implicitly the value of the last statement.)

Line 4 invokes method *major=* with the parameter "Computer Science" which is the sole argument of the *major=* function. Thus coding is simplified based on the method naming conventions which seem unusual to non-Ruby programmers.

3.2 Inner Classes

The Java programming language permits an inner class to be defined within an enclosing class. Nested classes were developed in Java to:

- logically group classes that make sense only in the context of an outer class;
- leverage encapsulation by both hiding the inner class as well as allowing the outer class's members to be private, yet accessible from the inner class;
- improve code readability and maintainability ("Java Tutorials")

Ruby implements inner classes, but in a manner much different from Java. Consider extending the registration system to include an inner class of a Section to manage information about a section's syllabus. An instance of a Syllabus would be created using the enclosing class.

```
mySyllabus = Section::Syllabus.new(textbook, topics,
                      assignments, policyStatement,
                      gradingStatement);
puts mySyllabus.textbook;
```

However, Ruby inner classes do not have access to the member of the outer class. Thus use of inner classes is less important in Ruby, and becomes more of an exercise in organization and taxonomy than one of functionality. Because Ruby can add methods dynamically, some of the traditional Java utilizations of inner classes can be accomplished more straightforwardly. Matsumoto explains, "Adding methods to objects can also be used in Ruby in situations where Java programmers use inner classes. For example, if you need to pass a listener object to some method, in Java you often instantiate an anonymous inner class that defines the listener methods and pass it. In Ruby, you can just create a plain object – an

instance of class Object – add the needed listener methods to it dynamically, and pass it" (Venners).

Eiffel does not support inner classes, nested classes, or anything that could be closely similar. There is, however, a small work-around to simulate what an inner class would do. By using the visibility techniques described in the Property Visibility section, one could make an external class behave as an inner class. If the ”inner class” were to define all of its features as only visible to one other class, only that other class would have the ability to use the “inner class” features. It would be similar to the following example:

```
class
   B
feature {A}
   f: STRING
   do
      Result := “Feature f”
   end
   g: INTEGER
   do
      Result := 6
   end
end
```

Class B defines two features only visible to A. So only A, the defining class (B), and any class that inherits from B can use these features.

There are no classes in IO; the distinction between class and instance does not exist. As IO is prototype-based, instance and class are unified. Oliver Ansaldi writes, “A prototype is both a class and an instance.” He continues, “To get a new instance, you clone an existing prototype. To get a new class you add or alter the behaviour of an existing prototype” (Ansaldi).

Even though Io is not class-based, it can emulate class-based programming. One example of this emulation is where the prototype mechanism encapsulates instance creation and initialization logic.

Whereas Smalltalk does not implement inner classes, it does contain a unique OO construct, called a block, that is much different than anything found in object oriented languages such as Java or C++. A block is actually more closely related to closures, lambda expressions, or nameless functions that can be found in languages such as Scheme and Python (Porter). In the Smalltalk workspace, which is simply a shell window, the following code can be entered and executed:

```
block := [:a :b | a + b].
block value: 3 value: 4.
```

The variable block is assigned the block of code in square brackets. First, *a* and *b* represent the parameters, which are denoted by the colon. The vertical bar then separates the parameters with the statements of the block. In the next line of code the block variable is sent the message #value:value:. This will return 7, by executing the statement within the block with the parameters 3 and 4, given in the message. A block can have from zero parameters, in which the message #value can be sent, up to as many parameters as desired. In the second case the message needs as many “value” arguments as expected by the block.

Blocks are actually very useful in Smalltalk because they are used for control flow as seen in the function for enrolling students or teaching assistants (which will also be discussed again in the section on multiple dispatch):

```
enroll: aStudent
   (aStudent class = Student) ifTrue: [
      (classSize < capacity) ifTrue: [
      “add student to the class”
      ]].
   (aStudent class = TeachingAssistant) ifTrue: [
      “add teaching assistant to the class”
      ]
```

For this function something different is expected to happen for Students and Teaching Assistants, so an if statement is checking the class of the function's parameter. An if statement consists of a Boolean expression, in parentheses, passed either #ifTrue:, #ifFalse:, #ifTrue:ifFalse:, or even #ifFalse:ifTrue:. Each of these messages takes a block as its parameter, as seen in the code above. In Smalltalk, if statements, as well as loops, depend on blocks.

4. Polymorphism

4.1 Method Overloading by Arity

Bertrand Meyer takes a dim view of method overloading and has stated, "the presence of overloading (is) a vanity mechanism that brings nothing to the semantic power of an O-O language" ("Significance").

Yukihio Matsumoto must have agreed with Meyer when creating Ruby. Ruby *does* allow you to define two methods with the same name in the same class; however, this does not provide overloading a method based on arity.

To test method overloading by arity, two different constructor methods for a class will be created – a common practice in C++ and Java.

In the Ruby example below, a class has two initialize methods to instantiate new objects. One method takes the instructor ID as a parameter, whereas the second method

does not.

```
def initialize(firstname, middlename, lastname, email,
               ssn, department)
   super(firstname, middlename, lastname, email, ssn)
   @department = department
end

def initialize(id, firstname, middlename, lastname,
               email, ssn, department)
   super(id, firstname, middlename, lastname, email, ssn)
   @department = department
end
```

A call to initialize with six parameters (the first signature) will fail with this error: `'initialize': wrong number of arguments (6 for 7) (ArgumentError).` The latter definition of initialize overrides the former, which is subsequently rendered invalid.

The only way to "fake" Ruby into method overloading is to define the method with *args, using the splat operator '*' to indicate an unknown or variable number of arguments, effectively creating a parameter which is an array.

```
def initialize(*args)
  if args.size == 6
     super(args[0], args[1], args[2], args[3], args[4])
     @department = args[5]
  elsif args.size == 7
     super(args[0], args[1], args[2], args[3], args[4],
           args[5])
     @department = args[6]
  else
     raise ArgumentError, "Initialization function takes
                           6 or 7 arguments only."
  end
end
```

While this is an interesting work-around, it makes for much less readable methods, as arguments are no longer named, and the developer must put exception handling in place to cover invalid arity.

Actual overloading of a method cannot be accomplished in Smalltalk (Sharp). The basic idea of constructor method overloading can be achieved, however, by simply creating as many class methods for instance creation as are needed. For example, two methods of different arity could be defined for the creation of a Student.:

- `#id:firstName:middleName:LastName:email:ssn:`
- `#firstName:middleName:LastName:email:ssn:`

Both these methods can be used to instantiate a Student, however the second function will not include the id of the instance being created. The code for the first message is as follows.

```
id: anIntegerID firstName: aStringFirstName
middleName: aStringMiddleName lastName: aStringLastName
email: aStringEmail ssn: aStringSSN
^self new id: anIntegerID firstName: aStringFirstName
          middleName: aStringMiddleName
          lastName: aStringLastName email: aStringEmail
          ssn: aStringSSN
```

This method returns a new instance of Student and sends it the message #id:firstName:middleName:LastName: email:ssn:. For a Student to understand this message, the method must also be included as an instance method in either the Student class or its superclass Person. The instance method will be denoted as private and will be used to set each of the instance variables. The same applies to the second class method mentioned above.

As an aside, note should be taken of the convention in which parameters are sent in messages. Smalltalk is dynamically typed and therefore it is convention to include the desired type in the variable name, such as aStringFirstName. This lets the programmer know that the first name should be a string (Hunt).

In order to demonstrate the instantiation of a Student object, let us define *s* to be a student with the following properties:

- id = 4
- firstName = ‘Kevin’
- middleName = ‘William’
- lastName = ‘Desmond’
- email = ‘desmon81@students.rowan.edu’
- ssn = ‘000-00-0000’

Student *s* could be instantiated by creating a new instance of Student and sending it the message #id:firstName: middleName:lastName:email:ssn:.

```
s := Student new id: 4 firstName: ‘Kevin’
     middleName:  ‘William’ lastName: ‘Desmond’
     email: ‘desmon81@students.rowan.ede’
     ssn: ‘000-00-0000’.
```

This is possible due to Smalltalk’s inability to have actual private methods. However, this is malpractice because the instance method was denoted as private. Instead, the word “new” should be dropped from this line of code. Now, the message is sent to the Student class instead of the instance s. The class method then creates the new instance and passes it the message.

There is no concept of a method definition in Io, but it is possible to create a method object and set it on a slot on an Object. If the method is first set to the slot with one argument, and then later it is set to a different method, only the second method will be attached to the object. Thus it is evident that there is no overloading by arity.

Considering Meyer's classification of overloading as a

vanity mechanism, it should not come as a surprise that Eiffel does not support this feature. Since features cannot have the same name due to the final name rule, no feature can have the same name. The objective of overloading constructors becomes possible only by defining multiple creation features with different names and parameters.

```
class
   A
create
   make, makeTwo
feature
   make
   do
      --Feature attributes here
   end
   makeTwo (amount:  INTEGER)
   do
      --Feature attributes here
   end
end
```

4.2 Method Overloading by Type

Another form of method overloading is based on the data types of a method's arguments.

As discussed in the previous section, method overloading in Io is impossible.

Method overloading by type is more feasible in Eiffel than overloading by arity would be. Again, Eiffel does not explicitly support method overloading but there is a way to circumvent this limitation. The generic class ANY offers the best way to mimic type overloading.

If an Eiffel feature is declared to take a parameter of type ANY, it is possible to convert the parameter to another type in order to preform manipulations. Similarly, features may return a variable of type ANY as well. All classes innately inherit from the ANY class ("Inheritance"). The only time this approach would not work, is if the different versions of the feature require a different number of parameters. In that case, one of the features would need to be declared with a different name.

In the registrar application, one requirement was to create a method that would automatically enroll a teaching assistant into a completely booked section, but not extend this same privilege to regular students. Hence the enroll method would need to behave differently depending on the type of the arguments.

The following example in Eiffel outlines the code necessary to implement the enroll feature of the SECTION class:

```
class
   SECTION
create
   make
feature
ta: TEACHING_ASSISTANT
student: STUDENT
enroll(x: ANY)
      do
         if x.same_type (ta) then
            update_enrollment(x)
            print("Teaching Assistant enrolled.")
            io.put_new_line
         elseif x.same_type (student) then
            -- Determine # of seats left in section
            if seats_left > 0 then
               update_enrollment(x)
               print("Student enrolled.")
            else
               print("Class is full. Student not
                      enrolled.")
               io.put_new_line
         else
            print(“The person is not a registered
                  student.”)
            end
         end
      end
```

As it can be seen from this example, SECTION has an instance of both STUDENT and TEACHING_ASSISTANT in the variables student and ta respectively. The enroll function accepts an input parameter of any type using the generic class ANY. The enroll feature type-checks the parameter x via comparison to its own internal instances of STUDENT and TEACHING_ASSISTANT to determine which statements to execute.

In Smalltalk, as done with the constructor methods, instance methods can be constructed with different arity, by creating several different messages. The registrar application required different formats to display an individual. One format can be modeled using the display method of Person (#display):

```
display
     ^ firstName, ' ', lastName, ' (' , email, ') '
```

To implement different formats for display, a new message #display: can be defined, which looks like this:

```
display: anIntegerFormat
   "display
     FirstName LastName (email) if anIntegerFormat = 0
    LastName, FirstName (email) if anIntegerFormat = 1
     email (firstName lastName) if anInteferFormat = 2"

    (anIntegerFormat = 0) ifTrue: [self display].
    (anIntegerFormat = 1) ifTrue: [^lastName, ', ',
                          firstName,' (' , email, ') '].
    (anIntegerFormat = 2) ifTrue: [^email, ' (' ,
                          firstName, '  ', lastName, ')']
```

It is not possible for any messages to have the same name. For example, the following method cannot exist in our program:

```
display: aBooleanFlag
   "if aBooleanFlag is false don't display email address"
```

```
(aBooleanFlag) ifTrue: [self display]
                       ifFalse:[^firstName, ' ',
                                lastName]
```

This is because the message #display: has already been used. It does not matter that they take different data types as parameters. Since Smalltalk is dynamically typed, the parameter types aren't determined until runtime. As far as the compilers are concerned, the two methods are defined identically. Therefore, we cannot really overload methods in Smalltalk, as mentioned in the previous section. A different approach must be taken to create this display method. The most obvious solution would be to give a different name, such as #displayWithEmail:.

As mentioned previously, method overloading by any mechanism is impossible for the similarly dynamically typed Ruby. To compensate for this, the parameters' types must be evaluated in a single function and conditional logic must then alter method behavior.

```
def display(param=false)
  # Use duck typing to see if param is boolean,
  # as there is no Boolean class in Ruby
  if !!param == param
    if param
      print . . .  # show email
    else
      print . . .  # do not show email
    end
  elsif param.is_a? Integer
    case param
    when 0
      print . . .  # using first format option
    when 1
      print . . .  # using second format option
    when 2
      print . . .  # using first format option
    else
      raise ArgumentError,
          "Invalid integer display parameter."
    end
  else
    raise ArgumentError,
       "Invalid type passed to display parameter."
  end
end
```

This absence of method overloading in Ruby necessitates the addition of extra code, not required in languages like Java.

4.3 Single Dispatch

Most object-oriented languages implement a form of object messaging known as single dispatch, where a message is sent to a receiver object (Chambers). Only at run-time will the type of the receiver be known; consequently the determination of which method to invoke is made during program execution.

Smalltalk is a highly polymorphic language. By sending data from one object to another via the use of messages, objects are capable of responding to any message they understand. For example, consider the method #display in the Person class which displays the name and email of that person in the form "First Name Last Name (email address)." Also, Instructors should have the word Professor prepended to the beginning of their name. It would be possible to write the following display method in the Person class:

```
display
   (self class = Instructor)
      ifTrue:  [^'Professor ', firstName, ' ', lastName,
                ' (' , email, ') ' ]
      ifFalse: [^firstName, ' ', lastName, ' (' , email,
                ') ' ]
```

This method determines the subclass of the Person and returns the appropriate way to display him or her. While more elegant, this does not demonstrate polymorphism in Smalltalk; therefore it is more reasonable to give the Instructor class its own method #display.

```
display
   ^'Professor ',firstName,' ',lastName,' (' , email,') '
```

When the display message is sent to an Instructor, this method will answer instead of the display message in the superclass Person.

The same technique of using different display methods in different classes was used in Ruby. The display method of Instructor introduces the variable *title* to prepend to an instructor's name. However, since TeachingAsst will inherit the method, a provision must be made to accommodate the case myTeachingAsst.display().

```
class Instructor < Person

 def display(param=false)
    title = (self.is_a? TeachingAsst) ? "" : "Professor "
```

The receiver of a single dispatch message is determined at run-time. The following Ruby code from class Test illustrates this principle using a *myStudent* instance of a Student object and a *myInstructor* instance of an Instructor object.

```
class Registrar

  def self.display(myObject)
    myObject.display(0);
  end

  if (gets.chomp.upcase == 'STUDENT')
    Registrar.display(myStudent);
  else
    Registrar.display(myInstructor);
  end
```

At runtime, the choice of the user will dictate whether

Registrar.display() will receive an instance of Instructor or Student. The call to myObject.display() will be dispatched to an object of a different classes depending on this choice. Ruby supports single dispatch, and the correct instance method is invoked at run-time.

Because Eiffel allows for multiple inheritance, it must allow for single dispatch in a more feature-rich manner than Smalltalk or Ruby. An Eiffel feature adaptation known as redefinition provides the ability to change the implementation of an inherited feature. If a TEACHING_ASSISTANT inherits a display method from both STUDENT and INSTRUCTOR, additional Eiffel clauses must clear up the ambiguity.

```
class
   TEACHING_ASSISTANT
inherit
   INSTRUCTOR
   STUDENT
      redefine display end
feature
   display
   do
      instructorDisplay := true
   end
end
```

Without the redefine subclause, the declaration of *display* would yield two features of the same name. This is made valid by specifying that the new declaration will override the old one. In a redefinition, the original version is called the *precursor* of the new version. It is common in Eiffel to rely on the precursor's algorithm and add some extra actions.

Prototype languages are highly flexible and Io is no exception. At runtime, there are many possibilities of what can change, including inheritance and the data defining the object. Io code consists of expressions, which in turn are comprised of message sends to objects which correspond to a function. Like most prototype languages, Io uses delegation to dispatch the correct method by following delegation pointers from an object to its prototype until a match is found.

4.4 Multiple Dispatch

In contrast to single dispatch, multiple dispatch will determine which method to use at run-time based on the types of the *parameters*.

Multiple dispatch is not a feature of Io. Perhaps the single most important reason for this design is that Io's creator, Steve Dekorte, refers to multiple dispatch as an "anti-pattern." He explains himself thus:

> Multiple dispatch makes the deep assumption that a notion of external type exists. But what notion of type can deal with something like a proxy, which is, in effect, all possible types?
>
> Any system that depends on a notion of type that is anything more than asking an object what it cho[o]ses to do at the moment violates encapsulation in ways that have serious consequences (Dekorte).

The above quotation gives some insight into Mr. Dekorte's distinctive programming language philosophy, which is reflective in the unique character of the Io language.

In June 2012, there was a lot of debate within the (small) Io community about whether multiple dispatch was needed. There were even doubts as to whether it is even possible to accomplish the task. During this debate, one Io developer was trying to refer Mr. Dekorte to the language Slate. Slate is also a prototype-based object language, yet is able to implement multiple dispatch. However, as Lee Salzman and Jonathan Aldrich point out, implemention of multiple dispatch is quite challenging in a prototype-based language:

> Such a combination (of prototypes and multiple dispatch) is difficult, however, because multiple dispatch depends on a predetermined hierarchy of classes, while prototypes generally allow a delegation hierarchy to change arbitrarily at any time (Salzman and Aldrich).

Eiffel does not support multiple dispatch innately. It is classified as a single-polymorphic language. Dynamic binding relies on the dynamic type of 'Current' only (i.e. the left hand operand in the case of binary operations). In Meyer's paper "Overloading vs. Object Technology," he stated his No-Overloading Principle: "Different things should have different names" ("Overloading"). Because of this principle, and the final name rule, multiple dispatch is ruled out. There have been some attempts at a work around for other languages, but nothing concrete was ever developed for Eiffel.

As discussed previously, Smalltalk is a dynamically typed language. Therefore it is not possible to define two methods #enroll: that take a different parameter, because they are considered the same at compile time. One possible solution would be to create two separate methods, #enrollStudent:, and #enrollTA:. Another solution, which will be used in our real example, is to check the class in which the parameter belongs. Since Students should not be added to a full class, Sections will also contain variables classSize and capacity. These will be compared before adding a Student to the class, but a

TeachingAssistant will be enrolled regardless of classSize.

```
enroll: aStudent
   (aStudent class = Student) ifTrue: [
      (classSize < capacity) ifTrue: [
      "add student to the class"
      ]].
   (aStudent class = TeachingAssistant) ifTrue: [
      "add teaching assistant to the class"
                          ]
```

Multiple dispatch is also not supported in native Ruby, as method overloading is not permitted. Similarly to the Smalltalk example, the class of the parameter needs to be checked, as in the code for the enroll method below.

```
# Enroll in a section
def enroll(pupil)
  case pupil
  when Student
    if ( (self.maxenrollment - self.students.count) > 0)
      updateEnrollment(pupil.id)
    else
      puts "This section is closed."
    end
  when TeachingAsst
      updateEnrollment(pupil.id)
  else
    puts "The individual is not registered."
  end
end
```

However, clever members of the Ruby community have developed language extensions to allow this behavior.

One such developer is Christopher Cyll who created a multiple dispatch library named "multi" (Cyll). Cyll illustrates how to create a method (e.g. "hiya") that will function differently depending on the parameter type.

```
 7 class Foo
 8   def initialize
 9     multi(:hiya, Integer) {|x| puts "Int: #{x}" }
10     multi(:hiya, String)  {|x| puts "Str: #{x}" }
11   end
12 end
13
14 f = Foo.new()
15 f.hiya(5)
16 f.hiya("hello")
```

When a new instance of Foo is created, the method multi() is called twice. The first parameter is the symbol :hiya. Ruby symbols represent names inside the Ruby interpreter (Thomas, Fowler and Hunt 631). The second parameter is an object type. What follows in braces is the method body.

Method multi resides inside library file multi.rb, where it performs the following process.

```
def multi(method_name, *patterns, &body)
  Multi::DISPATCHER.add(Multi::Dispatch, self,
                        method_name, patterns, body)
end
```

Dispatcher.add will dynamically define a method that is appropriate for the type of parameter that was passed. Ruby pattern matching is used to determine whether the parameter was an integer or string as shown in lines 15 and 16.

4.5 Operator Overloading

Operator overloading allows operators, such as + to have user defined meaning on user defined types. It is possible to conceive of an addition method for the user-defined class Course where the course number would be incremented by an integer value. C++ allows operator overloading as seen in the code below.

```
// Definition
Course Course::operator+(int rhs)
{
   int newCoursenum = this->coursenum + rhs;
   Course newCourse = Course(0, subject,
                             this->coursenum + rhs);
   return newCourse;
}
```

This example adds a special C++ member-function on the class Course to allows the addition of an int on the right hand side of the binary arithmetic operator +. Per this definition, the following become legal C++ syntax:

```
Course myCourse = Course(1,"CS",101); // Call constructor
Course newCourse = myCourse + 1;
```

whereas

```
Course newCourse = 1 + myCourse;
```

would not be.

In Ruby, many operators are actually method calls ("Programming Ruby"), as is the case with the + "operator." Therefore operator overloading simply requires the operator method to be defined. Obviously class Course has no such default + method defined.

```
  myCourse = Course.get(5)
  puts "#{myCourse.subject}-#{myCourse.coursenum}"
  myCourse = myCourse + 1
  puts "#{myCourse.subject}-#{myCourse.coursenum}"
```

The code above would produce the following result:

```
CS-4315
test.rb:99:in `<class:Test>': undefined method `+'
for #<Course:0x2a19cc8> (NoMethodError)
        from test.rb:16:in `<main>'
```

However, with the addition of the + method in Course as shown below, adding the integer 1 to this course results in "CS-4316."

```
  def +(x)
    self.coursenum += x
    self
  end
```

Similar to Ruby, operators such as +, -, *, and /, are binary messages in Smalltalk. The code myCourse := myCourse +

1 can be executed as long as Course can understand the message #+. Otherwise the error "MessageNotUnderstood Course >>+" will appear. The method #+ can be added to the class:

```
+ anIntegerX
        courseID := courseID + anIntegerX.
            ^self
```

With the addition of this message, Courses can now understand the #+ operation.

In Eiffel, as in Smalltalk and Ruby, operators are method calls defined as aliases. For example, a + b * c is short for a.plus(b.product(c)). Incrementing an instance of COURSE would not work natively as the expression "course := course + 1" is undefined for the type COURSE. Defining an alias to associate '+' with COURSE would appear as in the following code.

```
class
   COURSE
feature
   plus alias "+" (other: like Current): like Current
      do
         courseNumber := courseNumber + 1
      end
```

Thus, if the original course is CS-4315, calling course := course + 1 (or similarly course := course.plus(1)) would result in the new course number CS-4316.

Io is no different from Eiffel, Smalltalk or Ruby. Generally, each operator (e.g., Io's three assignment operators `::=`, `:=` and `=`) is actually a method on an object. The language web-site explains, "These operators are compiled to normal messages whose methods can be overridden" (Buday). Developer Ben Nadel explains further, "overriding an operator simply requires overriding an existing method (slot) on the given object" (Nadel). Operator overloading is yet another example of the dynamic nature of interaction between methods and slots.

5. Abstraction

5.1 Abstract Classes

An abstract class is a class that cannot be instantiated. Its properties and methods are only available via concrete subclasses.

Again, there are no classes in Io. Yet the characteristic of abstraction is found in Io. Darren Broemmer states that the "primary abstraction mechanism in Io is objects created through prototypes" (Broemmer). Oliver Ansaldi explains further, "To get a new class you add or alter the behaviour of an existing prototype" (Ansaldi). This is another example of how a feature is not explicitly in Io, yet the characteristics or spirit of a feature is present in Io.

Eiffel supports the notion of abstract classes, but refers to them as *deferred*. A class is deferred if it has at least one deferred routine. If a routine is declared as deferred, it means there is no implementation of the routine in that class.

Deferred classes cannot be instantiated objects, but it is possible to assign an instance of a non-deferred descendant to a variable of a deferred type. In the Registrar example, PERSON would be declared as the deferred class. There is not one general algorithm to display the information of a person, thus each of its descendants all implement a different way to display their information. Since the display feature would be a deferred feature, the class itself would be a deferred class.

```
deferred class
   PERSON
feature
   display
      deferred
      end
```

Since PERSON is a deferred class, no instance of it can ever be created, but INSTRUCTOR and STUDENT can still inherit from it. Each of these two classes and TEACHING_ASSISTANT will all have their own implementation of the display routine. The display routine defined by each of the descendants is said to be *effective* instead of *deferred*. It is important to note that there is a difference between effecting a feature and redefining it. A redefinition of a deferred feature would only take place if it was necessary to change the signature of the feature (Attapattu).

Each class in Ruby can have local variables (visible only in methods), global variables (which can be changed from anywhere by anything), class variables (an object instance can change the value for the entire class), instance variables and methods.

In the class registration model, when an instance of a Student object is created in Ruby, it inherits Person's instance variables (e.g., "@firstname, @email") and methods (e.g., "display()"). This instance will call its initialization super method to invoke the accessors of its parent object, Person. Person's initialize method, as well as its accessors, are available to be called from *any* class. There is no way to limit the invocation of the constructor method Person.initialize and Person's accessors to only the subclasses Student and Instructor.

Java would easily handle this requirement by defining Person as an abstract class. Ruby does not allow for that

level of abstraction. Could the constructor method be "protected" in the Java sense? Unlike Java, Ruby-style inheritance does not determine a method's visibility. For instance, a subclass can access a superclass's private methods as well as its protected methods (Flanagan and Motsumoto 232).

However it is possible to block initiation of the Person class by raising an exception if the caller is a Person.

```
def
initialize(id,firstname,middlename,lastname,email,ssn)
    if !self.instance_of?(Person)
   # instantiate instance variables
    else
      raise NoMethodError, "Initialization impossible;
                    Person treated like abstract class."
    end
  end
```

Smalltalk also does not explicitly support abstract classes. However, a developer could design any super classes to be implicitly used as an abstract class. In the Registrar System example, the Person class will be *implicitly* abstract. Although the Smalltalk compiler will not give errors, anyone developing this system should simply never create any instances of Person and respect the wishes of the class author! Normal rules for class hierarchy apply in Smalltalk, therefore, any instance variables or methods of the person class can be accessed by any of its subclasses.

6. Conclusion

6.1 Observations about each Language

6.1.1 Smalltalk

One unique aspect of Smalltalk is the class browser. The creating and editing of all classes is done within this graphical user interface. Different Smalltalk IDE's have slight variations in their class browser. The figure below shows the class browser for Pharo.

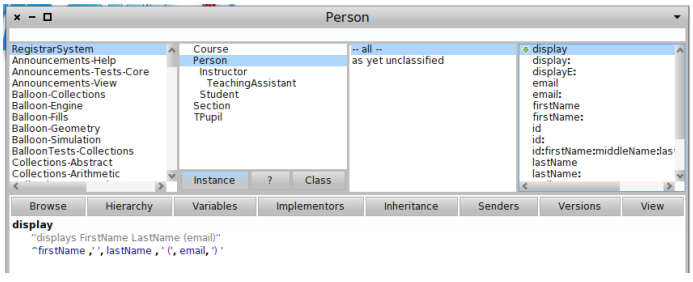

Fig. 4 Pharo: A Smalltalk class browser

When the IDE is started, the developer is prompted to choose an image file. This image file contains all the classes built into Smalltalk, as well as classes unique to the IDE with which it was installed. Once the class browser is opened, the classes contained in the image file are shown, and the developer can start adding their own classes to the image. In Pharo's class browser, the first column represents a list of packages. The second column is the names of all the classes in that package. Class names are indented to show their hierarchy. Methods can be grouped together in categories such as getters and setters. The names of the categories will appear in the third column. The last column displays the names of the methods. The bottom half of the window is where code is typed.

In general, the class browser is very helpful for organization. In many other languages, large programs can become difficult to read. With the class browser, a developer can easily navigate through the code.

The fact that Smalltalk trusts its programmers is a difficult concept to grasp. It seems undesirable that many aspects of Smalltalk, such as abstract classes and property visibility are not implemented explicitly. There were many times while developing the registrar system, where ignoring the fact that some methods should be private seemed like the simpler thing to do. However, it was important to use good programming style and not "cheat".

6.1.2 Eiffel

Eiffel, in general, was an easy language to understand. The syntax is very clean and even still partially used today in various forms of UML diagrams. There are several useful features in and about the language itself.

The first of these features is the downcast operator. This operator provides the ability to cast an object of any type down to a descendent of the object. This operation does require runtime checking to perform.

```
myNumber: INTEGER
myString: STRING
example(x: ANY)
do
        if x.same_type(myNUMBER) then
                myNumber ?= x
        else
                myString ?= x
        end
end
```

In this code segment, a feature named "example" is defined and takes a parameter of type ANY. Inside of the feature, a type check is done on the parameter passed. If the parameter happens to be an integer, the condition will be true and the if statement will execute, otherwise the parameter is a string. The '?=' operator will downcast the value passed in as x to whichever variable type matched it. The only downside to this operator is it can only cast down, not up. So x ?= myNumber would result in an error

since ANY is the parent of INTEGER.

Another feature of Eiffel doesn't really add any extra functionality to the language, but instead simplifies the effort needed to learn and understand the language. There are only two types of loop constructs in Eiffel. The first of these two loops is the across loop. This loop is used mainly for iterations over a list of items and is very similar to a for-each loop in many other languages.

```
across
        myList
as
        ic
loop
        print(ic.item)
end
```

In this example, we are looping over a list named "myList" and printing each item. The local entity ic is an instance of the class ITERATION_CURSOR which provides the access to the elements in the list. There are two other forms for this kind of list which can be obtained by changing the word "loop" to either "all" or "some" (universal and existential quantifiers, respectively) that provide the use of boolean expressions.

```
across
        myList
as
        ic
all
        ic.item.count > 2
end
```

In this loop, the code inside the "all" portion is true only if all items in the list have counts greater than two. Similarly, if "all" were to be replaced with "some," it would now only be true if at least one item's count is greater than two. This form of a list was introduced in January 2010 ("Meyer's Blog").

Eiffel's original loop syntax is more simplified and can still perform similarly to the "across" loop. However, the original loop can also mimic while and for loops that would be seen in other languages.

```
from
        myList.start
until
        myList.off
loop
        print(myList.item)
        myList.forth
end
```

This loop example mimics the same for-each loop in the first example of the "across" loop. The 'from' condition is the starting point for the loop – in this case the start of the list. The 'until' portion is the condition in which the loop will terminate – in this case when there is no current item in the list. The call to my_list.forth during the loop execution will move the current cursor position in the loop to the next item.

Beyond all the goodness Eiffel brings, there are some downsides to the language. The most distinctive flaw is the in the creation of an object. There are currently two ways to instantiate an instance of a class. The first method is uses a line of code such as `!!ta.init`, which will create an object of type TEACHING_ASSISTANT. The other way is shown in the code create `myList.make(5)` which, will create a list with the count of 5. The former approach is not as descriptive as the latter, but neither is very clear as to what is being instantiated. This is ironic in a language that is partially known for its descriptive nature.

6.1.3 Ruby

Ruby is a nice clean language whose syntax is easy to grasp. It has several unique and interesting features. One useful construct of the Ruby language is the inclusion of two forms of the logicals for *or, and* and *not*.

```
!                           Not
<=, <, >=, >                Comparison
==, !=, =~, !~ (etc.)       Equality / pattern
&&                          Logical "and"
||                          Logical "or"
=, +=, -=, ||= (etc.)       Assignment
not                         Logical negation
or and                      Logical composition
```

Fig. 5 Ruby operator precedence (Thomas, Fowler, Hunt 339)

In this partial list of operator precedence shown above, the assignment operators have lower preference than `!`, `&&` and `||`, but lower precedence than *`not`*, *`or`* and *`and`*. This allows code such as:

```
#  Cancel a section.
#  Default campus to Main in case of DB nulls
if affectedCampus = Section.get(mySecNum).campus or
   affectedCampus = "Main"
      updateCancellationList(affectedCampus, mySecNum);
end
```

Assignment has priority over the logical composition operator *and*. Thus, the if statement will assign a value to variable affectedCampus from the database information. Assuming the database has a value for campus, the if statement will evaluate to true, short-circuit and proceed to update the cancellation list for that campus. In the event the database has a null value for campus, the assignment will evaluate to false and the default value "Main" will be assigned to affectedCampus. This type of short-circuiting would not be possible with the `||` operator, as its precedence is higher than `=`.

While short-circuiting is a valid use of the logical composition operators, less useful is mixing these

operators with !, && and ||. It is confusing that, while *&&* has higher precedence than ||, *or* and *and* have the same precedence, and are therefore processed by left associativity.

This does allow the creation of some parenthetically devoid expressions, such as

```
s.status == "Undergraduate" && s.major == "Mathematics"
or s.status == "Graduate" and s.major == "Mathematics" ||
s.major == "Computer Science"
```

which will check to see if a person's status is either an undergraduate math major, or a graduate student in math or computer science. Yet this code is less readable than a Boolean expression containing helpful parentheses.

Ruby also has highly flexible case statements. Normally, case statements limited the when clause to equality operations as in the example below.

```
case grade
  when "A"
    puts "Great work";
  when "B"
    puts "Good job";
  else
    puts "Try Harder";
end
```

Behind the scenes, Ruby is using the case equality method === to evaluate each when clause, e.g., grade==="A". But as shown in the operator overloading section, === is just another method that can be overloaded to create unique case statement behavior.

```
# Define some constants
HONORS_COURSE = Course.null()
HONORS_COURSE.subject = "HONR"
PSYCHOLOGY_COURSE = Course.null()
PSYCHOLOGY_COURSE.subject = "PSY"
CHEMISTRY_COURSE = Course.null()
CHEMISTRY_COURSE.subject = "CHEM"
BIOLOGY_COURSE = Course.null()
BIOLOGY_COURSE.subject = "BIOL"
ASTRONOMY_COURSE = Course.null()
ASTRONOMY_COURSE.subject = "ASTR"
PHYSICS_COURSE = Course.null()
PHYSICS_COURSE.subject = "PHYS"

def ===(comparedCourse)
   self.subject == comparedCourse.subject
end
```

In the class Course, we can establish several constants of type Course – each of which will have only their subject assigned. When paired with the new definition of Course case equality, interesting case statements can now be written.

```
puts case myCourse
 when Course::HONORS_COURSE
   "Honors courses require a 3.3 cumulative average."
 when Course::PHYSICS_COURSE, Course::CHEMISTRY_COURSE,
      Course::BIOLOGY_COURSE
   "#{myCourse.subject}-#{myCourse.coursenum} might
   count toward the lab science requirement.  See your
   advisor."
 when Course::ASTRONOMY_COURSE, Course::PSYCHOLOGY_COURSE
   "#{myCourse.subject}-#{myCourse.coursenum} cannot
   count toward the lab science requirement."
 else
   # No special message needs to be printed
end
```

However one of Ruby's most distinctive features is called "duck typing" after the popular expression "If it walks like a duck and talks like a duck, it must be a duck." The principle for duck typing is that an object's type is determined by what it can do, not by its class. (Thomas, Fowler and Hunt 370).

The program below illustrates duck typing. In class Duck, we define a duck as something that quacks and waddles. There are also two other classes: a hypnotized person who has been programed to quack and waddle like a duck, and a normal person who might waddle from time to time when intoxicated, but never quacks.

```
 1 class Duck
 2   def quack
 3     print "\"Quaaaack!\" "
 4   end
 5   def waddle
 6     print "waddle, "
 7   end
 8 end
 9
10 class HypnotizedPerson
11   def quack
12     print "\"Kwack\" "
13   end
14   def waddle
15     print "waddle without embarrassment, "
16    end
17 end
18
19 class NormalPerson
20   def waddle
21     print "waddle self-consciously, "
22   end
23 end
24
25 def inDanger duck
26   duck.quack
27   for i in 1..3
28     duck.waddle
29   end
30   for i in 1..2
31     duck.quack
32   end
33 end
34
35 daffy = Duck.new
36 julia = HypnotizedPerson.new
37 will = NormalPerson.new
38
39 puts "\n\nDaffy is in danger."
40 inDanger daffy
41 puts "\n\nJulia is in danger."
42 inDanger julia
43 puts "\n\nWill is in danger."
44 inDanger will
```

In this program, When a duck is in danger, it will quack once, waddle three times, and then quack twice more to alert its flock. The inDanger method expects to receive a duck, as it will expect the passed object to quack and waddle. The output for this program is listed below

```
Daffy is in danger.
"Quaaaack!" waddle, waddle, waddle, "Quaaaack!"
"Quaaaack!"

Julia is in danger.
"Kwack" waddle without embarrassment, waddle
without embarrassment, waddle without
embarrassment, "Kwack" "Kwack"

Will is in danger.
duck.rb:26:in `inDanger': undefined method `quack'
for #<NormalPerson:0x24c2b80> (NoMethodError)
        from duck.rb:44:in `<main>'
```

Fig. 6 Output from Ruby duck typing

Duck typing was used in the registrar application to evaluate whether a parameter was a Boolean type. As Ruby does not have a class for Boolean (it only has classes TrueClass and FalseClass), duck typing was used to see if a parameter walked like a Boolean and talked like a Boolean.

```
if !!param == param
```

The Not operator !, if applied twice to a Boolean variable will equate to the original variable. This is the behavior of Booleans, so duck typing compensates for the lack of a Boolean class.

Another useful feature is the shortcut to avoid explicit creation of getters and setters via the attr_accessor statement:

```
attr_accessor :courseID, :subject, :coursenum, :title
```

There is not much missing in Ruby. It is understandable, in a dynamically typed language, why method overloading by type is not permissible. However, method overloading by arity was a nice way to reuse a method name with different signatures. Also, 0 does not evaluate to false, so the puts statement in the code below would unexpectedly execute.

```
myInt = 0
if myInt
  puts "myInt is true"
end
```

6.1.4 Io

Io was an interesting language, and much different than the other three languages evaluated. Io is a very flexible language that is simple, yet powerful. It has ingrained meta-programming features and is appropriate for exploratory software creation.

6.2 Final Thoughts

Today, of these four Languages, Ruby is clearly the most popular, especially when coupled with Rails, a web applications framework written in Ruby. Ruby on Rails is used for many popular websites such as Hulu, Urban Dictionary, and Groupon. The web sites range from traditional, content-based sites to online radio sites, to web hosting sites.

Smalltalk, though not as popular as Ruby, has had some application in the business world. For example, JPMorgan has used Smalltalk to implement Kapital, an advanced financial risk management and pricing system. The Orient Overseas Container Line (OOCL) uses Smalltalk for their Integrated Regional Information System, IRIS-2. This system coordinates all the information the company and its employees use.

Eiffel has seen some scattered use, but its popularity is waning. It has been described as "one of those academic languages no one uses." It suffers from weak tools; it is "so sophisticated" that good Eiffel tools are harder to write. Eiffel lacks a well-organized open source community.

Io is not as popular in the business world, but is still common in open source and personal projects. The Io community is very small, fractured, and fraught with disagreements.

In an agile application development environment on the web, Ruby would be a good choice. As it borrowed many of its conventions from Smalltalk, it may end up being a better language choice in most situations.

Eiffel, however, would be the best choice if one wanted to model a complex object-based system. It is not surprising that a recent use of Eiffel at AXA Rosenburg Investment Management was for a knowledge management system. Its ability to model abstract concepts would make it a strong choice for such an application.

Io is in a class by itself, despite the fact that it has no classes. (Pardon the pun.) But as a recent prototype based language, it might be useful to model artificial intelligence based systems. While it would be exhausting in AI development to build a comprehensive class inheritance model, the highly dynamic nature of Io might lend itself well to a very ad hoc encapulsation of the real world. It is more likely that the human brain classifies in terms of real life examples, than in a predetermined taxonomy. However, Io might not be the best choice of a prototype language. Newer languages such as Acute or Slate may show more promise and may gain the all-important critical mass needed to invest heavily in a programming language.

Works Cited

Alpert, Sherman R. "Primitive Types Considered Harmful." Java Report 3.11 (1998): n. pag. Primitive Types Considered Harmful. IBM. Web. 29 Oct. 2012. <http://www.research.ibm.com/people/a/alpert/ptch/ptch.html>.

Ansaldi, Olivier. "Blame It on Io! A Slow-paced Introduction to the Io Language." Ozone: Software Development Bits'n'Bobs. N.p., 15 Mar. 2006. Web. 27 Oct. 2012. <http://ozone.wordpress.com/2006/03/15/blame-it-on-io/>.

Attapattu, Nayanajith Shehan. "12.1 Using an Eiffel Deferred Class: COMPARABLE for Eiffel Book." Scribd. N.p., n.d. Web. 19 Nov. 2012. <http://www.scribd.com/doc/56926465/79/Using-an-Eiffel-deferred-class-COMPARABLE>.

Broemmer, Darren. "On the Applicability of Io, a Prototype-Based Programming Language." N.d. MS. The George Washington University. The George Washington University, 2007. Web. 30 Nov. 2012. <http://www.seas.gwu.edu/~mmburke/courses/csci210-f07/iolang.pdf>.

Buday, Andriy. "Developer's RoadMap To Success: Io Programming Language." Developer's RoadMap To Success: Io Programming Language. N.p., 8 July 2012. Web. 27 Oct. 2012. <http://www.andriybuday.com/2012/07/io-programming-language.html>.

Chambers, Craig. "Object-oriented Multi-Methods in Cecil." Proceedings of the European Conference on Object-Oriented Programming (ECOOP '92) (1992): 33-56. Springer-Verlag. Web. <http://www.springerlink.com/content/0q07t07138t53004/>.

Cyll, Topher. "Multiple Dispatch." Ruby Forge. Ruby Central, 22 Dec. 2005. Web. 29 Oct. 2012. <rubyforge.org/projects/multi>.

Dekorte, Steve. "[Io] Keyword Arguments." *Io Language Group*. Yahoo! Groups, 14 June 2012. Web. 27 Nov. 2012. <http://tech.groups.yahoo.com/group/iolanguage/message/13235>.

Flanagan, David, and Yukihiro Matsumoto. The Ruby Programming Language. North Sebastopol: O'Reilly, 2008. Print.

Hunt, John. Smalltalk and Object-orientation: An Introduction. London: Springer, 1997. Print.

"Inheritance." Eiffel Documentation. N.p., n.d. Web. 20 Nov. 2012.

Iolanguage.com. Io Online Community, n.d. Web. 13 Oct. 2012. <http://www.iolanguage.com/>.

"Io Programming Guide." *Io Programming Guide*. N.p., n.d. Web. 23 Nov. 2012. <http://iolanguage.org/scm/io/docs/IoGuide.html>.

"The Java Tutorials." *The Java Tutorials.* N.p., n.d. Web. 07 Nov. 2012

Meyer, Bertrand. "Bertrand Meyer's Technology Blog." Bertrand Meyers Technology Blog RSS. N.p., 26 Jan. 2010. Web. 30 Nov. 2012. <http://bertrandmeyer.com/2010/01/26/more-expressive-loops-for-eiffel/>.

Meyer, Bertrand. "Invitation to Eiffel - ISE Technical Report TR-EI-67/IV." *Eiffel Software*. Interactive Software Engineering Inc, July 2001. Web. 29 Oct. 2012. <http://archive.eiffel.com/doc/online/eiffel50/intro/language/invitation-00.html>.

Meyer, Bertrand. Object-oriented Software Construction. 2nd ed. New York: Prentice-Hall, 1997. Http://scholar.googleusercontent.com/scholar?q=cache:eYD4SEL68Z8J:scholar.google.com/+criteria+for+object+oriented+language&hl=en&as_sdt=0,31. Web.

Meyer, Bertrand. "Overloading vs. Object Technology." Journal of Object-Oriented Programming, Oct. 2001. Web. 18 Nov. 2012. <http://se.ethz.ch/~meyer/publications/joop/overloading.pdf>.

Meyer, Bertrand. "The Significance of .NET" Eiffel Software. N.p.,n.d. Web. 29 Oct 2012.

Nadel, Ben. "Seven Languages In Seven Weeks: Io - Day 2." *Seven Languages In Seven Weeks: Io - Day 2*. N.p., 30 Nov. 2010. Web. 30 Nov. 2012. <http://www.bennadel.com/blog/2066-Seven-Languages-In-Seven-Weeks-Io-Day-2.htm>.

Porter, Harry H. III. "Smalltalk." Smalltalk Overview. Portland State University, 24 Mar. 2003. Web. 25 Nov. 2012. <http://web.cecs.pdx.edu/~harry/musings/SmalltalkOverview.html>.

Programming Ruby – The Pragmatic Programer's Guide. *Programming Ruby – The Pragmatic Programer's Guide*. Addison Wesley Longman, Inc. , 2001. Web. 07 Nov. 2012. <http://www.ruby-doc.org/docs/ProgrammingRuby/>.

Sadeh, Benny, and Stephanne Ducasse. "Adding Dynamic Interfaces to Smalltalk." *Journal of Object Technology*. ETH Zurich, May-June 2002. Web. 23 Nov. 2012. <http://www.jot.fm/issues/issue_2002_05/article1/index.html>.

Salzman, Lee, and Jonathan Aldrich. "Prototypes With Multiple Dispatch: An Expressive and Dynamic Object Model." (2004): n. pag. Carnegie Melon University. Web. 27 Nov. 2012. <http://www.cs.cmu.edu/~aldrich/papers/ecoop05pmd.pdf>

Scharli, Nathanael, Stephane Ducasse, Oscar Nierstrasz, and Andrew P. Black. "Traits: Composable Units of Behaviour." N.p., 2003. Web. 27 Oct. 2012. <http://scg.unibe.ch/archive/papers/Scha03aTraits.pdf>.

Sharp, Alec. Smalltalk by Example: The Developer's Guide. New York: McGraw Hill, 1997. Online. <http://stephane.ducasse.free.fr/FreeBooksByExample/>.

Stewart, Bruce. "An Interview with the Creator of Ruby." An Interview with the Creator of Ruby. O'Reilly Media, 29 Nov. 2001. Web. 26 Oct. 2012. <http://linuxdevcenter.com/pub/a/linux/2001/11/29/ruby.html>.

Stroustrup, Bjarne. "Why C++ Is Not Just an Object Oriented Programming Language." ACM SIGPLAN OOPS Messenger 6.4 (1995): n. pag. Why C++ Is Not Just an Object-oriented Programming Language. Association for Computing Machinery. Web. 26 Oct. 2012. <http://dl.acm.org/citation.cfm?id=260207>.

"The Eiffel Programming Language." Eiffel. University of Michigan and University of Maryland, 26 Nov. 1996. Web. 17 Nov. 2012. <http://groups.engin.umd.umich.edu/CIS/course.des/cis400/eiffel/eiffel.html>.

Thomas, David, Chad Fowler, and Andrew Hunt. Programming Ruby: The Pragmatic Programmer's Guide. Raleigh, NC: Pragmatic Programmers, 2005. Print.

Venners, Bill. "Dynamic Productivity with Ruby: A Conversations with Yukihiro Matsumoto, Part II." *Dynamic Productivity with Ruby.* Artima, Inc., 17 Nov. 2003. Web. 07 Nov 2012. <http://www.artima.com/intv/tuesday.html>.