Why Java Isn't Smalltalk: An Aesthetic Observation

Bad Movie

I watched a bad movie the other night. It doesn't matter which one; we've all seen bad movies, and we recognize one when we see it: one in which the acting is too broad, the lines are repetitive, sequences that are supposed to be humorous or moving drag on too long until you either fidget or reach for the fast-forward button on the remote control, and the sight gags are obvious and overdone. These are movies that are, in short, aesthetically unpleasing.

Furthermore, the running critique in the viewer's head as s/he catalogs yet another of the movie's flaws serves to distract from the movie's original purpose: its plot. When the movie ends, the viewer may not even remember the plot, but will certainly remember all of its aesthetically unpleasing elements!

To the experienced Smalltalker, Java is like a bad movie. It, too, is aesthetically unpleasing. Most of us have been striving all of our Smalltalk careers for some ephemeral quality we call "elegance" in coding and design. It is a quality that could also be called "aesthetics." We do not seek this quality because we are all really artists in programmer's clothing (though some of us are!), but because our experience has shown us that programs that are more elegant are easier to create, to understand, and to maintain than those that are not.

Elegance – What Is It?

Though we talk about elegance a great deal in the Smalltalk community, and we are quick to recognize elegant designs when we see them, we very rarely stop to define exactly what we mean by the word. This is probably because it is a difficult word to define – even my Random House College Dictionary (revised edition) wasn't much help. It's definition of "elegance" was rather vague, but it did lead me through a chain of synonyms which I believe capture the essence of what we as programmers mean when we talk of elegance:

> Elegance: "something elegant; a refinement"
> Refine: "to bring to a fine or a pure state; to purify"
> Pure: "free from anything of a different, inferior, or contaminating kind…unmixed"

Simplicity in Smalltalk

At its core, then, elegance is captured by purity, sameness or simplicity. These are qualities which are inherent in the design of the Smalltalk language – in its syntax, in its short lists of reserved words and operators, and in its basic tenet that "everything is an object."

When we teach beginning Smalltalkers, we tell them that its syntax is one of the simplest of all programming languages, consisting of the form:

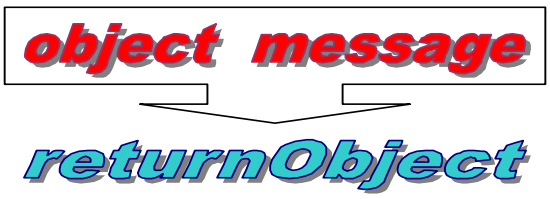


Figure 1: Smalltalk Syntax

All actions in Smalltalk are realized by sending messages to objects, and all messages return an object. Since everything in Smalltalk is an object, everything understands some set of messages. Furthermore, since even classes in Smalltalk are objects, even the creation of new instances is achieved by sending messages to objects.

We further teach our students that Smalltalk has only five reserved words (true, false, nil, self and super) and two operators (assignment and return) – short lists to remember. Other than these reserved words, anything else can be

declared as a variable. Other than these two operators, everything else they encounter will be either an object or a message, following the same rules of syntax as all other objects and messages.

This is simplicity, purity, refinement. It is elegance. It is aesthetically pleasing.

Complexity – what is it in Java?

A quick look at Java shows that it is not nearly as simple. Its syntax is sometimes – but not always – object-followed-by-message; its lists of reserved words and operators are long; methods don't necessarily return objects; all objects-like things are not necessarily objects; and all class-like things are not necessarily classes. Furthermore, data and behavior hiding are subjective, exception handling is rigidly enforced, classes can be declared to either always be superclasses or never become superclasses, and some objects may change the types of other objects in a way that forces the programmer to change the types back.

Was That a Reserved Word I Saw You With Last Night?

The list of reserved words in Java 1.0 takes up half of a page in *Java in a Nutshell*. There are fifty-nine of them – plus nine more reserved method names. These are long lists of words to remember to avoid using as variable names and message names – much longer than Smalltalk's single list of 5 reserved words.

When is a Java Object Not an Object?

Unlike in Smalltalk, everything in Java is an not object. There are "real" objects – those that inherit from the root Object class, and primitive data types (integers, characters, floating point numbers, Booleans, etc.). The latter, since they are not objects, cannot be sent messages, cannot serve as superclasses for other classes, and cannot have their behavior redefined.

Among Java's messages, not all return objects. As a side-effect of Java's strong typing, the return type of a message must be declared, and it can be declared to be an object type, a specific primitive data type, or nothing (void).

…Or a Message Not a Message?

While most of Java's syntax consists of the form:

```
object.message();
```

there are notable exceptions to this rule. Instances, for example, are not created by sending messages to objects; rather a reserved word, *new*, is used in conjunction with the name of the class which – for the purposes of creating instances only – takes the form of a message name. And the *new* keyword comes before the "class/method name", rather than after it like other messages. Java's "constructors," as they are called, look like this:

```
new Classname();
```

Due, in part, to the existence of its non-object primitive data types, not all actions in Java are achieved through message sends. Rather, there is a list of some forty-four operators that may be performed – some appropriate for one or more subsets of the primitive data types, some for objects, and some for both. Since these are not messages, they do not fit within the *object.message()* format; rather, they precede, follow, or fit between one or two operands. Furthermore, they cannot be "overloaded" (applied to objects/types other than those defined for the language) – a process we call "polymorphism" for their Smalltalk cousins.

Of Java's operators, one of the most confusing is *instanceof*. Most of the operators are composed of one or more special characters, which could serve to make them relatively easy to distinguish from messages, but *instanceof*, as a collection of alphabetic characters, can appear to the uninitiated as a message. It is not; it must fit between two operands like the other binary operators.

…Or a Class Not a Class?

This is only the tip of Java's complexity iceberg. It gets worse. Not all class-like things in Java are really classes. Some are interfaces and the two constructs behave differently: classes are extended, and interfaces are implemented. While I will admit that the interface notion is a powerful one – and is certainly less complex than its C++ counterpart (multiple inheritance) – it adds yet another complexity to the language.

Visibility Modifiers

In Smalltalk, data and behavior visibility are simply and consistently enforced: all data is private, and all behavior is public. Some might claim that the latter, particularly, is an arbitrary language-design choice, and point out that Smalltalkers have struggled to overcome the shortcoming of all behavior being public with either conventions in method comments or tools that separate public methods from private ones. They will further argue that, even then, the language fails us by *still* allowing any object to send the so-called "private" messages to our objects. All of this is true. A quick look at Java's visibility modifiers, however, will show that Smalltalk's consistency in defining what is always public and what is always private makes it a simpler language.

In Java, there are five visibility modifiers, all of which can apply to either data or methods (collectively called "members" in the Java vernacular). The five modifiers are: public, "default" (also referred to as "package"), protected, private-protected, and private. These modifiers determine which objects can directly access the data or call the message – as well as which objects inherit the data or the method – according to whether the accessing or inheriting object resides in the same package as the object from which it is attempting to access or inherit a member.

How the five visibility modifiers cut across the packages according to accessibility and inheritance would require a three- or four-dimensional space to diagram (in order to map the modifier against the membership in a package against whether access is allowed against whether inheritance is allowed) but I'll attempt to describe it in words. First the two easy ones: predictably, public members are accessible to every other object and inherited by every subclass of the object, and private members are available only to the defining object. Private members cannot be inherited, either by subclasses within the defining object's package or those outside of the package. Protected members can be accessed only by objects in the same package, but can be inherited by objects in other packages. Private-protected members are accessible by no object other than the defining object, but can be inherited by subclasses in any package. Finally, the default visibility (package), allows members to be inherited and accessed by objects within the defining object's package, but neither inherited nor accessed by objects outside the package.

Try-Catch

The Java compiler insists that any method that calls another method that throws an exception must either 1) handle the exception or 2) declare that it, too, throws the exception (or, alternatively, the calling method could handle part of the exception and throw the rest). While most Smalltalk dialects also provide exception handling, it is not in any way enforced by the Smalltalk compiler. Exception handling is, by its very nature, a complex addition to a program that requires the programmer to think about the implications of the actions of his object on a scope far outside that object. This has the potential to break – or at least severely bend – encapsulation, since one object must know how another object is implemented enough to know what exceptions it may throw and how it might handle those exceptions. The Smalltalk programmer has the luxury of thinking about exception handling relatively late in the development process, while Java forces the programmer to consider this complexity early – potentially, as often as with every method send.

Class and Method Modifiers

Not all classes in Java can become superclasses. Java allows a programmer to define a class or a method as "final." Final classes can never become superclasses, and final methods cannot be overridden. Later, if another programmer thinks of a legitimate extension to a final class, s/he is out of luck. Rather than create a new subclass with all the functionality of the superclass plus the new functionality, s/he must re-implement the entire class. Although most programmers would probably not create new classes assuming they would never become superclasses, final classes do appear in the language. Many of Java 1.1's "wrapper" classes, for example – classes designed to allow primitive data types to be "wrapped" in and treated as real objects – are declared as final.

Other Java classes *must* be superclasses. These are the classes declared as "abstract". Abstract classes cannot be instantiated. (This makes abstract classes like interfaces, which are by their very nature abstract – whoops! except in Java 1.1, where interfaces can be instantiated, adding a layer of inconsistency onto another inconsistency….), Furthermore, any class that inherits from an abstract class must either override all of the superclass' abstract methods and provide concrete implementations for those methods or (if it has even a single abstract method) must also be declared abstract.

Methods in Java may also be "static." Static methods are like Smalltalk's class methods in that they cannot access the class' instance variables, but can refer to the class' static variables (which are like class variables). The catch is that static methods are also implicitly final, so they cannot be overridden.

These class and method modifiers, while making it explicit what can and cannot be overridden and/or subclassed, add another complexity to the Java language.

Casting

Another obvious difference between Java and Smalltalk is the issue of static vs. dynamic typing. Both have their strengths, and I won't argue here that Smalltalk's dynamic typing is somehow objectively "better" than Java's static typing. There does exist, however, at least one impact of Java's static typing on the language's complexity: the need to cast objects into different types.

Some general-purpose Java classes, like its Vector class (Java's version of OrderedCollection), take as arguments objects of any Object type (i.e., any subclass of Object, which excludes primitive data types). In order to accomplish this within the context of Java's strong typing, the object being added to the vector must be cast to the highest point in the inheritance hierarchy: the Object class. As a result of the upward cast, when the object is later retrieved from the Vector, although it is still an *instanceof* the original class, it can only respond to messages defined for the Object class. If the programmer needs to send the retrieved object a message defined for that object's class, s/he must cast the object back down to its original type. Not all casting is obvious. Upward casting (like to the Object class) is implicit; downward casting must be explicit.

Impact of Complexity

Perhaps by now I've convinced you that Java is a more complex language than Smalltalk. So what? Big deal, right? Aside from being aesthetically unpleasing to the Smalltalk programmer, is there any real, measurable impact of a language's complexity on a business' bottom line? In a word: yes. There are two important consequences of this complexity: 1) on training costs and 2) on programmer productivity.

A Juggling Act

G. A. Miller, in his 1956 paper *The Magic Number Seven +/- 2*, concluded that a person's short-term memory is limited to holding onto approximately seven unrelated ideas at a time. Beyond an approximate maximum of 9 ideas, one or more items drop out of short-term memory to make room for the new idea.

At one time or another I think we have all, as programmers, experienced the sensation of feeling like a juggler, keeping approximately seven "balls" (unrelated ideas or concepts) in the air (our short-term memory) as we attempt to pull them together into a coherent whole. If a colleague walks up to your desk when you are in the middle of your juggling act – even if he says nothing, even if the interruption is brief – the distraction is enough to cause you to let most (if not all) of the balls fall to the floor.

The distraction, however, does not have to be external; it can be the result of having to figure out some sub-section of your code in order to solve a larger problem. A C-programmer friend of mine says, for example, that he has to do this every time he calls the printf() function. He has to go look up which special characters are used for which purposes, and what the format of the string should be.

This is a mental process that could be called "subroutining" – something computers are very good at. Programming languages are designed to put one context on the stack in order to switch to another context while a sub-section (a subroutine) of the code executes.

The problem for humans is that our short-term memories do not contain stacks. As Miller shows, we have an approximately seven-item "scratch pad". When we switch contexts, all of those approximately seven things that we were trying to relate to one another *must* be dropped on the floor in order to make room for another seven things from a different context. If the context switch is of a short enough duration, it may be possible for us to catch the original balls on the first bounce and reconstruct their relationships to each other in our short-term memories. Longer duration context switches, however – one subroutine that leads to another and another and another – decrease the

possibility that the original seven ideas will be reconstructed in the same manner as they were originally. Putting the seven balls back into the air in the same configuration as they were before the first interruption is an unlikely and error-prone prospect.

The more complex a programming language is, the more frequent these context switches will be. For example, when faced with a business problem to solve, the Smalltalker will ask the following hierarchy of questions:

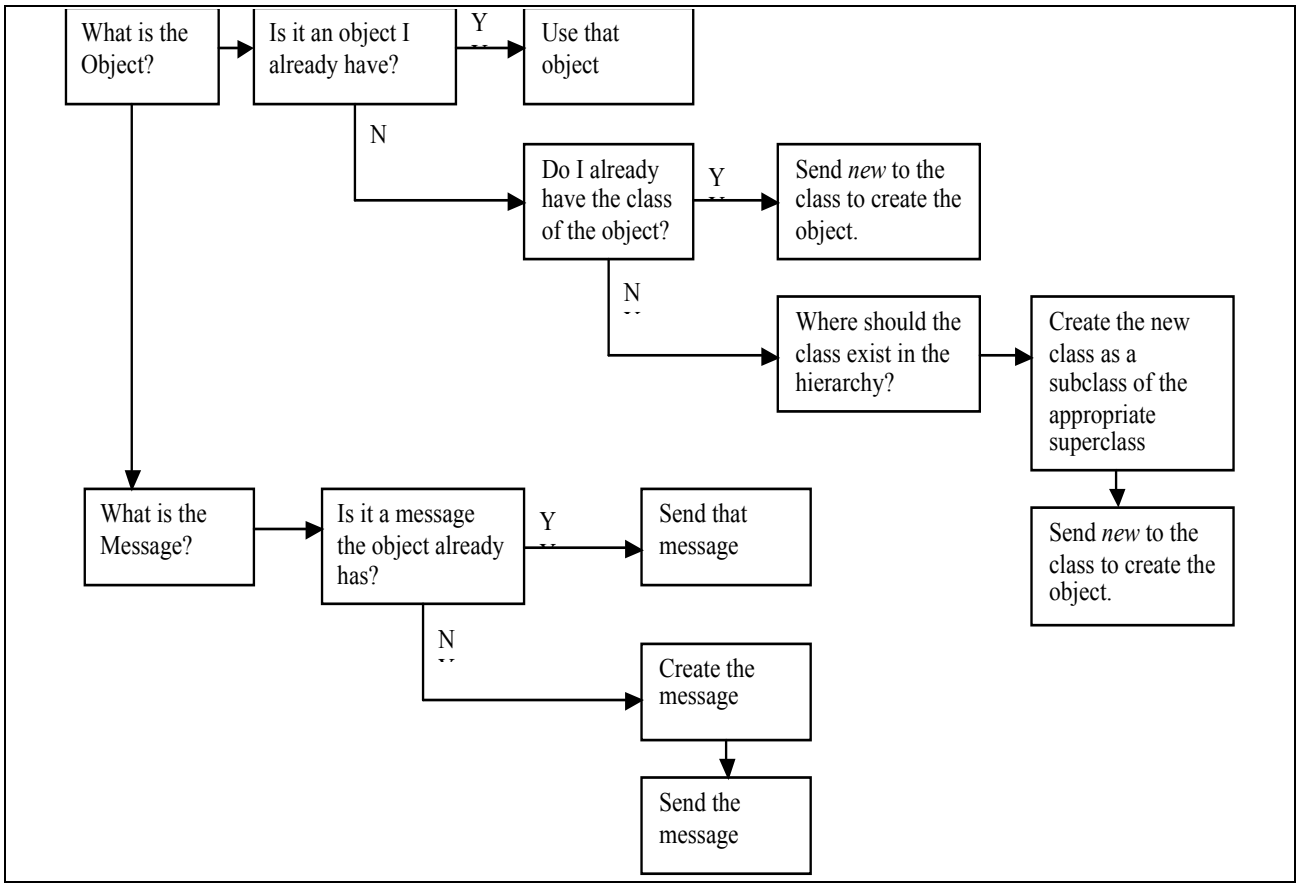


Figure 2: "What are the Object and Message?" Smalltalk Decision Tree

By contrast, the Java subroutining may look like this…

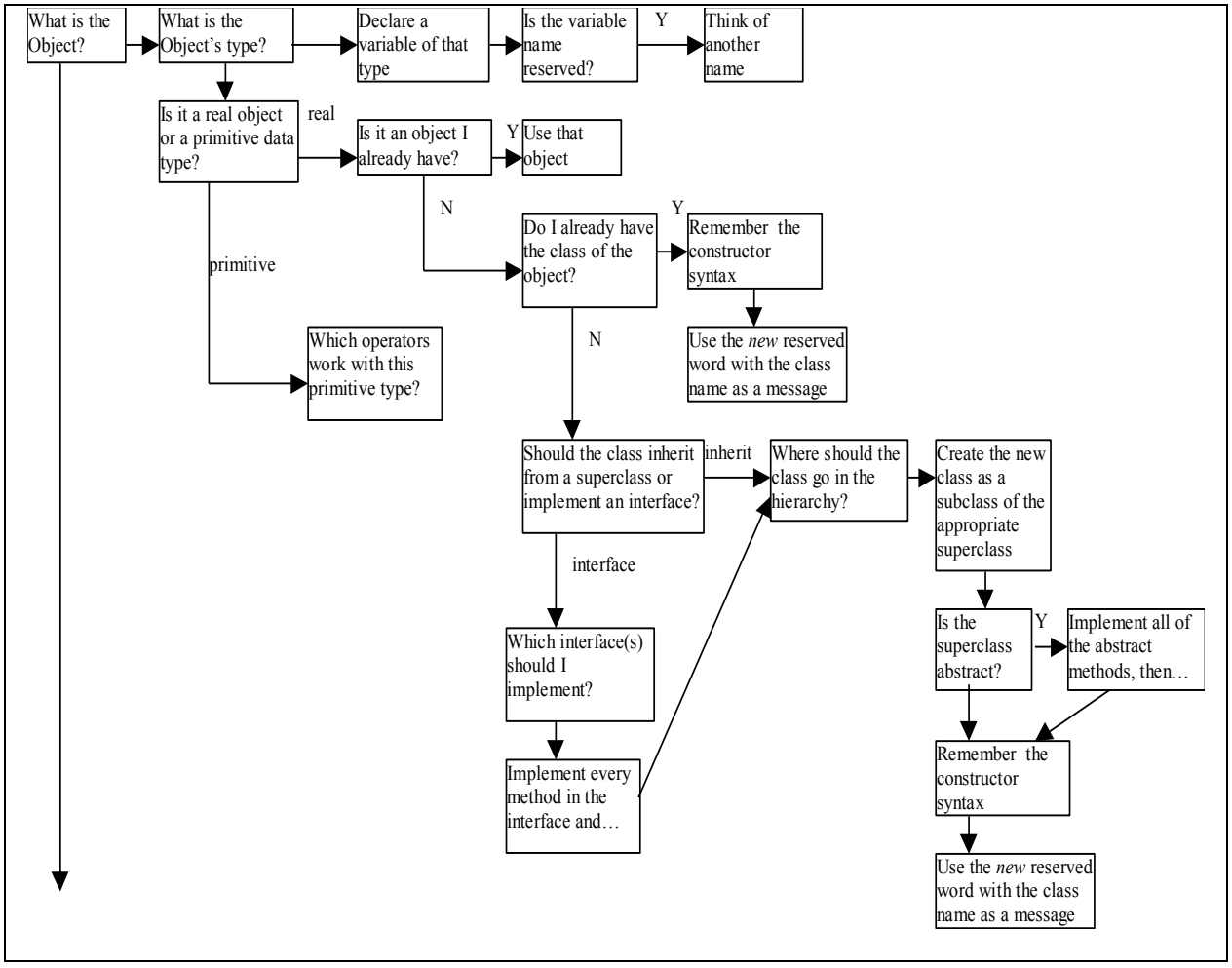


Figure 3: "What is the Object?" Java Decision Tree

…and this…

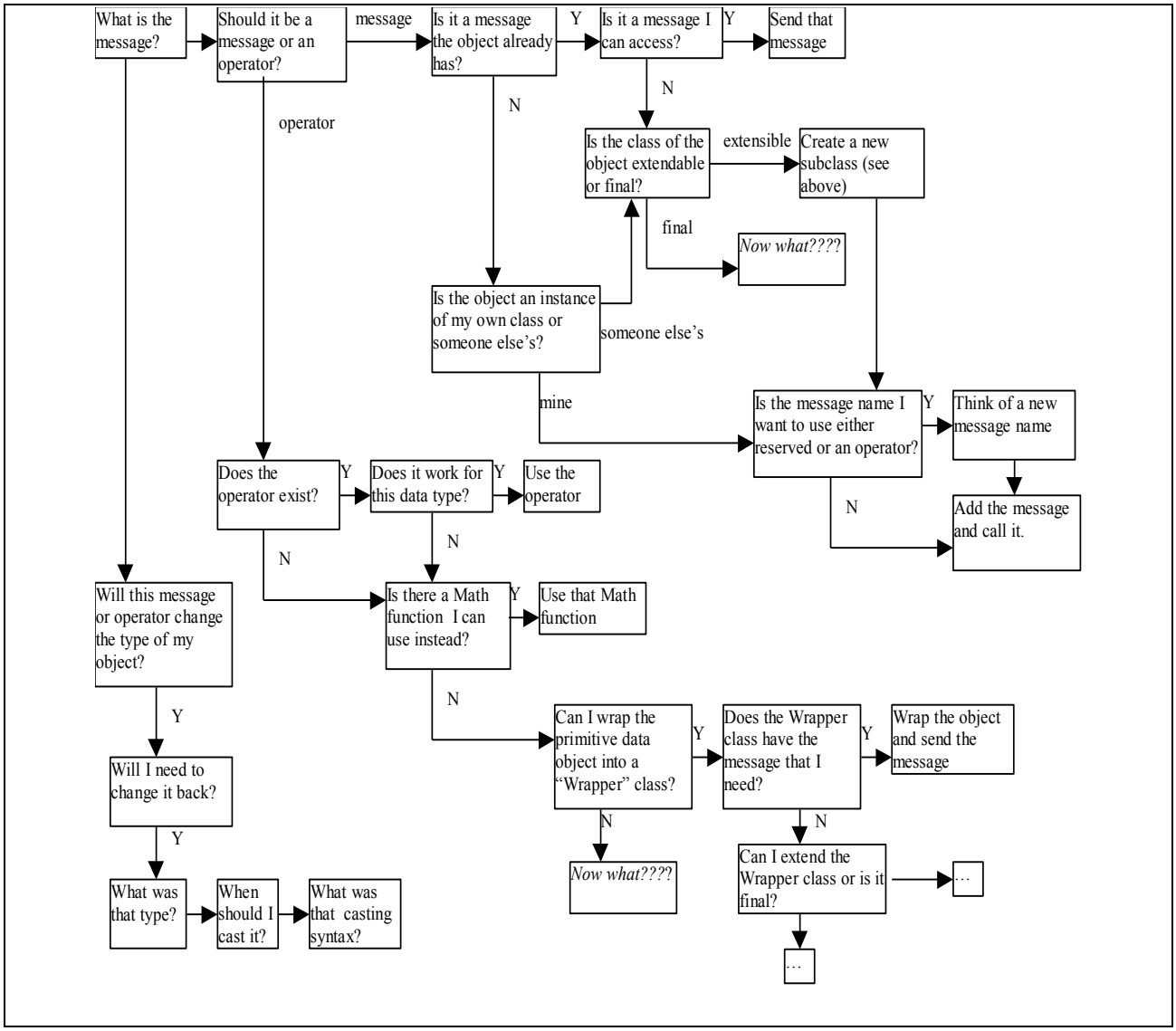


Figure 4: "What is the Message?" Java Decision Tree

Obviously, Java's greater complexity requires more subroutining, and this extra subroutining will take more time and be subject to more errors than a simple subroutine. And this is just the complexity of sending a single message to a single object. It says nothing of reading pre-existing code – a necessary evil if any maintenance is required – where you must decide if what you're reading is an object, a primitive, a message, an operator, or one of those 59 reserved words!

"Chunking" – Automatic Subroutining

In spite of the differences in relative complexity in Smalltalk and Java, I am not claiming that Java is so complex that it will be impossible for anyone to ever become proficient in the language! We all know from experience that, given sufficient time any thinking process – even a complex one – will become automatic. How is this possible?

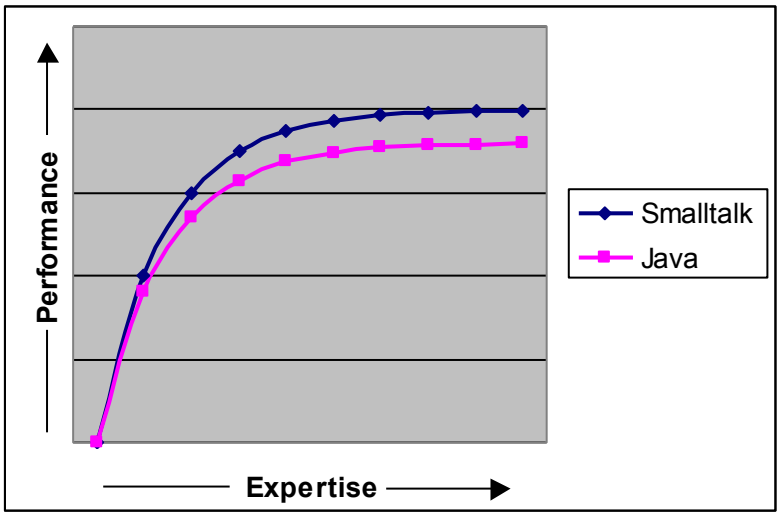
As Miller pointed out in his *Magic Number 7,* our short-term memories are limited to approximately seven items only if those seven items are unrelated. Learning is the result of what Miller calls "chunking": relating previously unrelated ideas in short term memory into one coherent "chunk" which can then be remembered as a unit. If we go through the same subroutine enough times, our brains will create a pattern out of the subroutine, so that we can eventually perform automatic subroutining. (These patterns are not unlike the popular "design patterns" of programming: recognizable groupings and interactions of classes that form a higher level of abstraction than a single class on its own.)

The problem with complexity is that the more subroutining a person must do, the slower will be the chunking process, which means learning the language will take a lot longer. There are more branches to visit in the decision tree, so each branch will be visited less often and each will be less likely to become part of a recognizable pattern, or chunk. By contrast, given a simple or consistent paradigm, it will be easier and faster to chunk something, because more things will chunk to the same pattern. A pattern that is used more often is more likely to be chunked.

Chess example? Example of chunking. A chess expert is more likely to chunk things he's seen lots of times; no one will chunk arbitrary chess layouts.

Furthermore, in a simpler language, it will be necessary to learn fewer *total* chunks to accomplish the same task. Therefore, if two otherwise equal learners – one in Smalltalk and one in Java – have an equal amount of time to learn their respective languages, each will create an equal number of chunks, but the complexity of Java will mean that the Java learner's collection of chunks will apply to less abstract problems (only a small subset of what could be accomplished), while the Smalltalker's collection of chunks will be more complete, more abstract, more powerful, and able to be applied to broader issues. A Smalltalker, for example, will be able to spend more time thinking about the modeling of the business domain while the Java programmer is still thinking about language constructs.

Graph 1 plots a Smalltalk programmer's performance (i.e., the speed with which s/he can produce a given unit of code, or the inverse of the number of mistakes s/he is likely to make) against a Java programmer's performance for a given amount of expertise (i.e., time to learn the language, number of chunks formed).



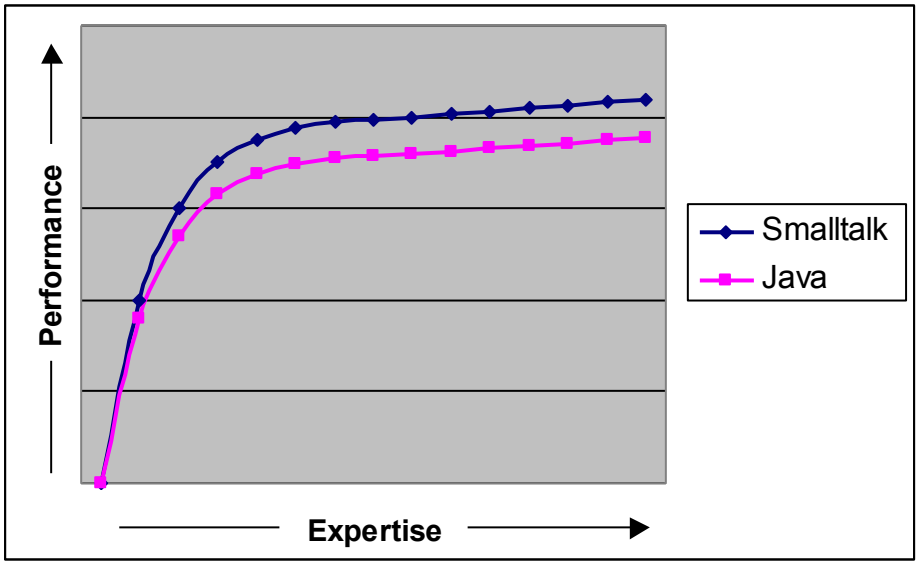
Graph 1: Performance for Java and Smalltalk Programmers Over Time

What About Those Experts?

If a programmer can *eventually* learn either Smalltalk or Java and become proficient in either language, perhaps a language's relative complexity is only an issue in the learning stages. If so, perhaps the language's complexity only impacts training time and training costs. Once a programmer becomes an expert, doesn't the graph level out to the same ideal of "infinite expertise?" After expertise is achieved, aren't all languages the same?

Actually, no.

In his famous study on cigar-rolling operators, [Crossman, 1959] showed that there really is no such thing as a quintessential "expert." As long as a person keeps performing the same task – even a task as seemingly trivial as rolling a cigar – that person will keep improving his or her performance. S/he will continue to learn, continue to "chunk", make fewer mistakes, and become more efficient at the task. There is no such thing as infinite expertise; the learning graph never levels off. If we were to extend Graph 1 out into time, it would look like this:



Graph 2: Continued Performance Improvement Over Time

Therefore, over time a Smalltalk programmer's performance will continue to exceed that of a Java programmer with the same amount of expertise. The Smalltalker will make fewer mistakes and produce results faster than the Java programmer, and this productivity will affect the business' bottom line. The Smalltalk program will have a shorter time-to-market; the Smalltalker's business will benefit from the quicker sale (or internal use) of the product; the Smalltalker's business will have won the competitive race to marketability.

Conclusion

I am not saying that the Java language does not have certain inherent advantages over other languages – even Smalltalk. Java's built-in security, its namespaces, and its web-enabling technology all afford it advantages for certain applications. Furthermore, Java's compiler checks will probably produce more robust code, with fewer runtime errors, than the same code produced in Smalltalk – at least for the first test.

The question is whether a Smalltalker's inevitably greater productivity in producing code will still provide enough extra time to allow that Smalltalker to test and remove the runtime errors from his or her code – with time to spare for adding extra functionality.

Similarly, Java's complexity – the mechanics of the language, the myriad subroutining that one must do in order to perform what in Smalltalk are the simplest of tasks – force the programmer to think *about* the language, and – just possibly – to miss the *purpose* of the programming process: modeling the business domain. It's not as easy anymore to translate business objects into virtual objects, so perhaps when the programming ends, the programmer will have stopped caring about the business objects, and will walk away only thinking about how complex the language is.

I will continue to learn more about Java because I enjoy challenges, because I find the language to be a great improvement over many of the other non-Smalltalk object-oriented languages, and because I am curious about its

capabilities and limits. Even so, I will always enjoy using Smalltalk: for its simplicity, its beauty, it's elegance and – above all – for its productivity. Perhaps it's not just an aesthetic issue after all.