Dylan Competitive Analysis

draft 13-Feb-95, Stoney Ballard, Apple Computer

This paper discusses the competitive position of the Dylan language against C++, Objective-C, and Smalltalk. It covers the features and properties that make Dylan a superior language for mainstream development.

For decades, Smalltalk has been known as the only pure object-oriented language. Since entering its modern form as Smalltalk-80 and its release by Xerox, it has become widely used for prototyping and, lately, commercial programming. Smalltalk's appeal comes primarily from its simplicity and the highly interactive development environment. Smalltalk has had little or no success in mainstream applications development primarily because it produces very large and slow applications.

Please note that since each vendors implementation of Smalltalk is different, I use the generic Smalltalk to mean the language in general, while Smalltalk-80 refers specifically to ParcPlaces version.

Objective-C was developed as a cross between C and Smalltalk. It combines the low-level data types and efficiency of C with the high-level object system of Smalltalk. This combination produced a surprisingly useful language, but ultimately it was defeated in the marketplace by C++, probably because Objective-C is proprietary.

C++ has become surprisingly popular among mainstream applications developers. The combination of familiar C concepts and a fundamental efficiency have proven to be a very powerful draw. Unfortunately, C++ is the most complex language in common use, both in its syntax and its semantics. Experts commonly warn against using many of C++s features because of the likelihood of programmer error. While C++s object-oriented concepts and strong type system are an improvement over C, the complexity introduced by these features has driven many programmers to look for an alternative.

Dylan's goal is to combine the purity of Smalltalk with the efficiency of C++, using compilation techniques developed by the Lisp community. The result is a language which is easy to use, supports very strong object-oriented structuring, and produces small and fast applications.

The following sections demonstrate Dylan's key advantages:

1. Dylan's syntax is more readable and maintainable than the competition.
2. Dylan's memory management is better than any of the competition.
3. A production Dylan compiler can produce code whose performance is

comparable to well-written C++.

4. Dylan is suitable for producing standard, shrink-wrapped applications, with deployed size on a par with C++ and Objective-C, and much better than Smalltalk.
5. Dylan has the best balance between practicality and incrementality, giving turn-around times measured in seconds, but with high efficiency code.
6. Dylan provides the high visibility and control for which Smalltalk is prized, but without sacrificing optimization.
7. Dylan is as pure OO as Smalltalk, but provides the efficiency of C++ without impairing abstraction.
8. Dylan provides a modularity scheme clearly superior to any of its competitors.
9. Dylan provides better interoperability than any other language, including C++.
10. Dylan's non-proprietary standard supports widespread adoption of the language.

Article Summary

---

1. Readability and writability (syntax)

Dylan was designed to be both easy to write and easy to read and understand. The syntax resembles a less verbose Pascal, but still easy for programmers unfamiliar with Dylan to understand the code. For example:

```
if (foo > 3 & foo < 5)
    print(transcript, foo is in range);
end if;
```

Smalltalk syntax is frequently derided as weird. The uniform message-passing approach extends even to its control structures, giving rise to constructs such as the simple conditional, which is written as a message **ifTrue:** sent to a Boolean object. For example:

```
        (foo > 3 and: [foo < 5]) ifTrue: [Transcript show: foo is in
range]
```

Objective-C looks like a cross between C and Smalltalk:

```
if (foo > 3 && foo < 5)
    [transcript show: foo is in range];
```

The brackets enclose expressions written in Smalltalk syntax. There is a significant amount of conceptual whiplash as the programmer switches between two syntax, even within a single expression.

C++ was designed to continue Cs tradition of using as few keywords as possible, resulting in an almost unparsable and unreadable language. Consider the following example from the *C++ Annotated Reference Manual*:

```
// assume S is a type and a is a variable
S x(int(a));    // declares x to be a function
S y((int)a);    // declares y to be an object of type S
S z = int(a);   // declares z to be an object of type S
```

Conclusion: Dylan's syntax is more readable and maintainable than the competition.

---

2. Memory management

As C++ becomes more widespread, and as programmers start using it in an object-oriented way, they are learning an old lesson: manual memory management is expensive, fragile, error-prone, and impairs the interoperability of separately-designed parts. Memory management is a global problem. Solving it locally leads to failures and leakage when communicating outside the subsystem.

Although the use of leak detectors like Purify has increased, any rearrangement of subsystems subsequent to the repair of leaks requires rechecking and replugging, sometimes by removing the earlier fixes.

Automatic memory management, also know as Garbage Collection or GC, has been part of Smalltalk, Lisp, and Dylan from their inception. There have been attempts to add it to C++ and Objective-C, but neither the languages nor the compilers have been designed to cope with the needs of a GC (primarily ensuring that every live object has a pointer to it).

Dylan, like Smalltalk, allocates objects in the heap. While more expensive than stack allocation (as is commonly done with C++), heap allocation eliminates the possibility that an object may become deallocated while some client retains a pointer to the object. However, Dylan's use of type declarations and the avoidance of pointer arithmetic allows a Dylan compiler to allocate objects on the stack or inside variables if the compiler can determine that its safe to do so. The straightforward semantics of Dylan makes this kind of optimization tractable. As Dylan compiler technology improves, we should be able to realize similar storage efficiency to C++, but do so in a safer and more flexible way.

Of course, automatic memory management substantially improves code quality by eliminating the 30-50% of C++ instructions concerned with memory management. This is, it not only deletes statements and destructors, but awkward designs necessary to cope with the need to explicitly delete everything at exactly the right time and place.

Conclusion: Dylan's memory management is better than any of the competition.

---

3. Performance of generated code

C++ has inherited Cs reputation for high-performance code. Unfortunately, many C++ programmers are finding that the use of some basic C++ features, such as constructors, destructors, and type conversions, frequently leads to surprisingly inefficient code. In addition, the manually deleting objects as they die is considerably less efficient than the use of a copying GC, such as is used in Dylan.

Smalltalk-80s virtual machine buys incredible portability between platforms by insulating the program from the details of each platform, including the user interface manager, the file system, and the instruction set. While this technique is valuable when the same compiled program must run on a variety of platforms, the result is a program that runs 10-20 times slower than a similar program written in C.

Its very difficult to make Smalltalk produce fast code even without the virtual machine. Although some research has been done on dynamically self-optimizing code, the lack of type declarations in Smalltalk remains a huge barrier to efficient compilation.

Although Smalltalk has a very reactive and powerful development environment, it is practically inseparable from the language. As with Lisp, this leads to a situation where applications built using Smalltalk can dynamically reconfigure themselves by adding or removing code and classes. The performance penalty for this flexibility is very high.

Objective-Cs performance is somewhere between C++ and Smalltalk. The control structures and low-level data manipulation is just C, and it is very fast. The object operations are just Smalltalk, but arent designed for a virtual machine. This buys a substantial performance improvement over Smalltalk. A well-designed Objective C program (meaning one that uses C types in performance-critical areas) is comparable in performance to C++.

Dylan is capable of producing very fast code. Although it is more flexible than C++, it is purposely less flexible than Smalltalk. As with C++, the development environment can be separated from the Dylan program under development, so that the compiler can assume that only certain well-defined sorts of changes can occur to a completed Dylan application. In particular, Dylan supports dynamically-linked libraries, which can add subclasses and new polymorphic methods. The Dylan language allows the programmer to specify which classes and which generic functions can be extended in this way. Only those will retain flexibility at the expense of speed. The compiler will optimize away the flexibility of everything else in the Dylan program.

The straightforward semantics of Dylan allows the compiler to determine the appropriate degree of flexibility at every function call, producing tighter code than C++. Changes to the program, e.g. adding or removing classes and polymorphism, potentially affect the compilation of all the code in the program. The automatic program-wide optimization by the compiler eliminates the need for the programmer to reoptimize the details of her program whenever large changes occur.

Conclusion: A production Dylan compiler can produce code whose performance is comparable to well-written C++, but without the manual optimization performed by C++ programmers.

---

4. Size of generated code

Although many programmers understand the benefits of dynamic languages such as Smalltalk and Lisp, there has always been a barrier to use in the size of the programs generated by these systems. No matter how quickly you can build an application, if its huge, it cant be distributed. The size of these applications results from two sources: the inseparability of the development environment from the application program, and the large quantity of functions and objects supplied as the basic run-time library. Dylan is intended to be implemented with a more traditional model where the development environment is a separate application.

The Dylan language has been designed to minimize the number of required run-time

facilities, relegating anything that is useful but not really necessary to separate, optional libraries. The functions and classes required by the language are necessary and sufficient to ensure a high degree of compatibility between separately-developed modules. C++s lack of any standard libraries is a serious impediment to reusing code.

The module and library system in the Dylan language has been designed to support separate compilation and dynamically-linked libraries. The use of shared dynamically linked libraries (DLLs) for the basic run-time library and the application framework, as is common in Windows programs, results in distributable applications that are much smaller than comparable C++ applications that use an application framework.

Dylan benefits from much of the research into Lisp compilers. Techniques such as CPS conversion can be used to produce code that is highly optimized, with such features as tail-recursion removal, closure optimization, and lifetime analysis. This results in generated code that is comparable in size to that generated by mainstream C++ compilers.

Objective-C programs are similar to C++ programs, with the exception that Objective-Cs large class library is similar to Smalltalk's or Dylan's. Objective-C has been used primarily on UNIX, where shared libraries arent an issue for the user. Since Objective-C uses a traditional text-file-based compiler, and not Smalltalk's environmental approach, its generated applications are considered ordinary in size.

One of the surprising results from C++ is the large amount of space taken up by the virtual function tables. It is not uncommon to see 20-30% of a compiled C++ program consist of just these tables. Both Smalltalk and Dylan use a dynamically optimizing dispatch system to minimize this space without impacting performance significantly.

Conclusion: Dylan is suitable for producing standard, shrink-wrapped applications, with deployed size on a par with C++ and Objective-C, and much better than Smalltalk.

---

5. Support for (enabling of) incremental development environments

Smalltalk's incremental development environment is its most attractive feature. Smalltalk is unusual in that it models the process of development as incremental changes to a memory image, in contrast to a more traditional view of development as editing source code. Smalltalk's approach is naturally incremental, but does not fit in well with modern approaches to software construction, where source-code control systems are used in conjunction with a disciplined build process to produce a product with known quality levels.

In contrast, C++ is as non-incremental as possible. Programmers cringe when faced with making a change to a header file, as its usually necessary to recompile the entire program when such a change is made. Although some work has been done to produce systems that recompile the least amount possible, it requires a very large and expensive development environment, e.g. Lucids Energize, to manage the intricate dependency web within a C++ program. The main culprit here is C (and C++s) macro system, which operates as a text processor, leaving very little alternative to recompiling

everything when a macro definition changes.

Objective-C falls in between these two extremes. It uses a traditional text-file model, so has the batch compilation feel and the dependency problems introduced by changing header files, but its object system is entirely dynamic, which eliminates any need to recompile clients when some class they use changes. In later versions of Objective-C, a declarative type system was introduced, which can provide higher performance and compile-time type checking at the cost of longer turn-around time while compiling.

Dylan programming follows the traditional source-text-editing model, but has been designed for incremental development by running the macro system at the parse-tree level. This allows the dependency web within a Dylan program to be easily calculated and maintained. In addition the polymorphic dispatch mechanism uses dynamic linkages, resulting in a minimal need for recompilation whenever even major changes, such as changing the layout of a class, occur.

Conclusion: Dylan has the best balance between practicality and incrementality, giving turn-around times measured in seconds, but with high efficiency code.

---

6. Debuggability

Dynamic languages support the best debuggers. The run-time safety and the flexible linkages within dynamic languages allow the programmer to modify running programs by adding or modifying functions, classes, and variables. In Smalltalk, writing a program and debugging it are completely interwoven, and use the same tools.

Dynamic languages are flexible, yet safe at run-time because all objects are tagged in memory with their type. This allows dynamic linkages to be checked for correctness as the program runs, eliminating the crashes and memory corruption that frequently results from the use of the equivalent void* declaration in C and C++. This early error trapping is invaluable for finding the precise cause of bugs, rather then attempting to deduce the cause of memory corruption in a C++ program, where the bug and the program crash are logically unconnected.

C++ and Objective-C debuggers are based on C debuggers, which have only the types declared for variables to guide them. Most C++ debuggers are unable to determine exactly what kind of object a pointer references. The most advanced ones use heuristics based on the objects virtual function tables, but fail for classes with multiple inheritance or without virtual functions.

Even if C++ debuggers could show the program state accurately, the lack of any dynamic linkages in C++ makes it practically impossible to do more than minor adjustments to the state of a running program. In frustration, some super programmers patch the machine code of their C++ applications by hand so that they can avoid a bug rather than suffer the time to correct their source code, recompile, relink, and run the program to the point of failure again (if it can be recreated at all).

Dylan has much in common with Smalltalk, in that the program state is obvious to the debugger and the programmer. While Dylan has fewer dynamic linkages than other

dynamic languages, an incremental compilation system for Dylan can still make patches to running programs by maintaining a complete dependency network, recompiling and reloading all affected definitions when the programmer makes a change.

This combination of run-time type information and incremental recompilation allows a Dylan development system to clearly debug and safely update optimized code in a running application.

Conclusion: Dylan provides the high visibility and control for which Smalltalk is prized, but without sacrificing optimization.

---

7. Object-oriented techniques

The basis of OOP is abstraction. OOP relies on inheritance, encapsulation, and polymorphism as techniques which support abstraction. Inheritance allows detail to be layered, separating the general from the specific. Encapsulation lets the programmer specify, and the language enforce the separation of the relevant from the irrelevant (the public from the private) in an interface. Polymorphism supports using the same code in many situations, either as text or as compiled code.

The degree to which a programming language supports OOP can be seen in a simple example: a function **hypotenuse** which takes two arguments, A and B, and returns the square root of the sum of their squares. In single-polymorphism language, the A argument is known as **self** or **this**.

In Smalltalk, the function would be expressed as a method on **Number**:

```
hypotenuse: b

    ^(self * self + (b * b)) sqrt
```

This code is pure OOP. It doesnt matter what kind of objects **self** or **b** are, so long as they respond to the **+**, ***** and **sqrt** messages, this will work both as source text and as a running program. Some people think this is excessive, that the idea of sending a **+** message to an integer doesnt correspond to their mental model of arithmetic. However, the resulting uniformity and simplicity is valuable when mixing numeric types. This function will work with integers, real numbers, fractions, or any other **Number** types, with only one compiled version. It captures the basic abstraction of hypotenuse without any irrelevant detail.

Objective-C has these abstract qualities for high-level objects, but not for low-level ones. The equivalent function in Objective-C might be:

```
object hypotenuse( object b ) {
    return [[[self times: self] plus: [b times: b]] sqrt];
}
```

This doesnt work on C types like **int** and **float**, but would work on any objects that implemented the **times:**, **plus:**, and **sqrt** messages. Separate C versions would be

needed for the basic C types, and they would need different names as well. This discontinuity sacrifices abstraction for performance.

In C++, you can write this function as:

```
type1 hypotenuse( type2 a, type3 b ) {
    return sqrt( a * a + b * b );
}
```

Unfortunately, you need to supply a separate version of this function for each possible pair of argument types, even though the source text of the function would remain identical (except for the return and argument type names). At least all the versions of this function will have the same name.

C++ provides templates as a language mechanism to capture this textual abstraction, but there are complications in their use. To quote [Stroustroup 91]:

> The use of two (or more) template arguments for functions taking a variety of arithmetic types can also lead to the generation of a surprising number of different functions and thus to a surprising amount of generated code.

We can see here that while C++ provides a great deal of textual abstraction, it is not reflected as abstract compiled code, leading to larger programs that must be recompiled for each use.

There are ways around this. If you defined a class hierarchy for numbers, and used pointers to these numbers in the operations, then you could define hypotenuse as:

```
number* hypotenuse( number* a, number* b ) {
    return sqrt( a * a + b * b )
}
```

This looks okay, unfortunately, it leaks like a sieve. All these operations that take and return pointers to numbers have to heap-allocate them, and the intermediate results need to be deleted. This results in something like:

```
number* hypotenuse( number* a, number* b ) {
    number* temp1 = a * a;
    number* temp2 = b * b;
    number* temp3 = temp1 * temp2;
    delete temp1;
    delete temp2;
    number* temp4 = sqrt( temp3 );
    delete temp3;
    return temp4;
}
```

And hope that all the callers of hypotenuse remember to delete the return value when theyre done with it. At least this code is abstract even in the compiled form, so long as the unstated requirement that this code own the pointers returned from the intermediate operations remains true.

C++ programmers dont actually write code like this they simply avoid abstraction in

many cases.

In Dylan, were back to the simplicity and abstraction of Smalltalk:

```
define method hypotenuse( a, b );
    sqrt( a * a + b * b );
end method;
```

The advantage of Dylan over Smalltalk is that additional, special versions can be written for the high-performance cases, such as:

```
define method hypotenuse( a :: <short-float>, b :: <short-float> )
=> result :: <short-float>;
    sqrt( a * a + b * b );
end method;
```

The only difference between this version and the version without type declarations is that the compiler can know that this latter version will be called only when **a** and **b** are both of type **<short-float>**, and therefore compile this version to be as fast as in C. Providing special versions for only the performance-sensitive cases eliminates the code explosion problem of C++ templates, preserves the abstraction in the compiled code, yet runs as fast as C for the special cases.

This technique relies on multiple polymorphism, which is missing from Smalltalk and Objective-C, and is present in C++ only statically (as overloaded functions). Dylan's use of dynamic multiple polymorphism eliminates essentially all conditional statements which test the type of an object. By moving type-testing into the polymorphic dispatch system, the compiler is free to optimize it away when it knows the argument types supplied by a caller.

Conclusion: Dylan is as pure OO as Smalltalk, but provides the efficiency of C++ without impairing abstraction.

---

8. Modularity

Smalltalk-80, Objective-C, and C++, each have a single global name scope. While the C-based languages can use linker tricks to provide libraries with hidden, internal globals, Smalltalk cannot. Objective-C and C++ also inherit Cs static declarator for variables, which are local to the file in which they are declared. These tend to be more trouble than theyre worth.

C++ relies on classes to provide modularity. It has an elaborate system of declarations that expose three levels of access. Unfortunately, different access restrictions cant be presented to different clients without using the cumbersome friend declarations. Even with all this, nearly all classes are globally known.

Recently a new feature of namespaces has been added to C++ in an attempt to control problems caused by class name collisions. This isnt an access-protection system, its just a way of allowing names to be qualified by their home namespace name whenever a name conflict occurs. In practice, this implicitly adds the name of the namespace to

all the global names declared within it, making them even longer.

In Dylan, classes do not establish scopes at all. Since methods in Dylan are not attached to the classes they name, they dont have any privileged access to the slots (instance variables) of the class.

Instead, modules are used to establish separate namespaces with import/export control. Not only does this eliminate global name collisions (names can be renamed as they are imported), but separate modules can provide separate interfaces to the same variables. This is simpler and more flexible than C++s categorization of access into public, private, and protected buckets.

Dylan modules also work with Dylan libraries to provide a simple and direct way of specifying the interfaces of separately-compiled libraries. Theres no need for any non-portable linker tricks to provide suitable library APIs. In addition, libraries can export multiple modules, each one providing a different level of access.

Conclusion: Dylan provides a modularity scheme clearly superior to any of its competitors.

---

9. Interoperability

Smalltalk-80 buys portability by using a virtual machine, which is very successful at making Smalltalk programs binary-compatible across platforms. Unfortunately, the price is a high degree of isolation from the native facilities of the platform. For example, ParcPlace Smalltalk-80 doesnt use the Macintosh Toolbox other than to copy its internal screen buffer to a Mac window. All the windows, buttons, etc. are drawn using Smalltalk code, producing a significant performance problem. In addition, Smalltalk doesnt automatically incorporate changes to the System, such as when System 7 changed the window title bars.

This isolation from the OS applies to all C libraries. In ParcPlace Smalltalk-80, you have to write special C interface routines and link them into the Smalltalk runtime program in order to access C libraries. Other Smalltalk's, such as SmalltalkAgents, are much less isolated from the OS and C libraries, but they deviate substantially from the Smalltalk-80 language.

Of course, Objective-C and C++ have perfect interoperability with the C interfaces of the OS and ordinary libraries. You still have to link external libraries into the program, but this is the normal way of operating.

Dylan is about half-way between Smalltalk and C++. While wrapper routines need to be created to intercede between the semantically dissonant worlds of Dylan and C, Apple Dylan provides a tool named Creole to automatically create these interface routines simply by reading C header files. In addition, the C libraries are dynamically loaded, making the use of existing C libraries and OS calls easier than in C or C++.

Another aspect of interoperability is the ability to support libraries within the language. Smalltalk-80, for example, doesnt do this at all. A Smalltalk program is simply a

memory image. It has no mechanism for splitting the image into separate parts. C and Objective C work nicely with their library mechanisms. Objective-C works particularly well since it uses ASCII strings to name operations theres never any problem with library versions as there is with C++.

C++ has the well-known fragile base class problem, where even additive changes, such as adding a virtual function to an interface class, requires all clients to be recompiled. The C++ compiler optimizes interfaces beyond what is useful, mapping virtual functions to small integers so the clients can use simple table indexing for polymorphism. Unfortunately, this optimization means that adding a virtual function changes the indexes of all the following virtual functions in that class and all subclasses. This has caused much trouble in the industry, with SOM, for example, being required to use a C interface to hide the virtual function calls from the clients.

Dylan gives the programmer control over this problem. Dylan supports cross-library inlining and optimization, but only for classes and functions declared to be sealed. When flexibility is more important than performance, the programmer can unseal a class or function. This allows dynamically loaded libraries, or the main program to add new subclasses or new polymorphic methods to the interface item. The Dylan runtime dynamically optimizes the polymorphic dispatch for unsealed functions, producing acceptable performance, but the result is a degree of power unmatched by any other language.

Yet another aspect of interoperability is the use of class libraries within the main program. Smalltalk-80 led the way here by providing a complete set of utility and container classes, as well as a numeric system and a GUI framework. For the most part, Smalltalk programmers could buy and use third-party classes within their programs because everything rested on this common class foundation that shipped with every Smalltalk development environment.

Objective-C inherited much of this common class library approach from Smalltalk. Its implementation on the NeXT machine added a rich UI foundation that became the hallmark of the NeXT platform.

Unfortunately, C++ has resisted specifying any standard classes. One of the reasons for this is that there are system-wide architectural issues that have to be decided on a program-by-program bases. Class libraries that do not follow these architectures, particularly in their management of memory, will not integrate easily. If the language were to specify standard classes, then it would also have to decide these issues, and C++ is too low-level a language to do that.

The closest thing to a standard class library for C++ is an application framework like Apples MacApp, or Microsofts MFC. These libraries are primarily concerned with the UI, and since that is frequently the focus of a program, these frameworks get to establish the class and memory architecture for all the programs that use them. Unfortunately, they are still weak in many important areas, such as container classes, and do not establish any kind of automatic memory management.

Dylan has learned from these examples. The Dylan language specifies a set of required classes for containers, numerics, and system structure (e.g. class <function>) that all

implementations are required to implement. These classes form a complete foundation, and together with the automatic memory management, and the high-level nature of the Dylan language, serve to make separately-written class and function libraries easily integrable.

Conclusion: Dylan provides consistently better interoperability and compatibility with independently-developed libraries and OS APIs than any other language, including C++.

---

10. Standardization

One of the most important aspects of a computer language is whether programs you write in one implementation can be compiled with another.

Smalltalk is by far the worst of the languages we are considering here. ParcPlace owns Smalltalk-80, so Digitalk decided to make a similar, but incompatible version. Smalltalk Agents is Smalltalk in philosophy and name only. Its even more deviant from the original Smalltalk-80. Its simply impractical to move Smalltalk code from one vendors environment to another.

Objective C is similar in that it is also a proprietary language. Fortunately, NeXT started with the StepStone product, but they have been evolving ever since. This is OK, since NeXT is the only major user of Objective C. Its basically a one-vendor language.

C++ will be a standard someday. Theres an ANSI committee working on standardizing the language, but only after loading many new features. The de facto standard for C++ has been the various releases of CFront from AT&T. These releases have been marked by substantial changes from one version to the next. The releases all have their own set of bugs as well. Every C++ compiler vendor has implemented a somewhat different version of C++, and has fixed the bugs that were important to them.

The result is that C++ programs that have to compile with several compilers end up with lots of macros that adjust the source code for each compiler individually. There is also still the problem of compilers that dont implement the latest features, such as templates and exception handling. Programs have to be designed with the least-capable compiler in mind.

Dylan is in a rather different position. Like Smalltalk and Objective-C, Dylan is a proprietary standard of Apple Computer. Unlike them, Apple has released the language as an open standard anyone can implement it, but they cant call it Dylan unless it passes Apples acceptance test. This approach keeps the language simple and consistent, yet avoids the proprietary language aversion that stunted Smalltalk and killed Objective-C.

Conclusion: Dylan provides a practical, non-proprietary standard required to support widespread adoption of the language.

Summary

Dylan is a modern, thoroughly object-oriented language. Smalltalk, Objective-C, and C++ were developed before object-oriented programming entered the mainstream. Dylan's designers have had the benefit of our accumulated experience with the advantages and pitfalls of these older OO languages. Dylan emphasizes efficiency, clarity, productivity, and flexibility, producing the most practical object-oriented language yet.

Stoney Ballard Apple Computer

Please send feedback to dylan-comments@cambridge.apple.com
Additional copies of this document as well as additional Dylan information (reference manual, FAQ, implementations, etc.) are available on the Internet:

WWW page http://www.cambridge.apple.com
anonymous ftp site ftp://cambridge.apple.com/pub/dylan
netnews group comp.lang.dylan

Back to Advocacy Page