This is an excerpt taken from the paper: **C++? A Critique of C++ and Programming Language Trends of the 1990's. 3rd Edition. Ian Joyner.** Paper was written in 1996.
I, Jaime Niño, added some comments in places for more elucidation to the students.
The full paper, with no added comments, is available via the 4501-course web page.

The Role of a Programming Language

A programming language functions at many different levels and has many roles, and should be evaluated with respect to those levels and roles. Historically, programming languages have had a limited role, that of **writing executable programs**. As programs have grown in complexity, this role alone has proved insufficient. Many design and analysis techniques have arisen to support other necessary roles.

Object-oriented techniques help in the **analysis and design phases**; object-oriented languages to support the implementation phase of OO, but in many cases these lack uniformity of concepts, integration with the development environment and commonality of purpose. Traditional problematic software practices are infiltrating the object-oriented world with little thought. Often these techniques appeal to management because they are outwardly organized: people are assigned organizational roles such as project manager, team leader, analyst, designer and programmer. But these techniques are simplistic and insufficient, and result in demotivated and uncreative environments.

Object-orientation, however, offers a better rational approach to software development. The complementary roles of a**nalysis, design, implementation and project organization** should be better integrated in the object-oriented scheme. This results in economical software production, and more creative and motivated environments.

The organization of projects also required **tools external to the language** and compiler, like 'make.' A re-evaluation of these tools shows that often the division of labor between them has not been done along optimal lines: firstly, programmers need to do extra *bookkeeping* work which could be automated; and secondly, inadequate *separation of concerns* has resulted in inflexible software systems.

C++ is an interesting experiment in adapting the advantages of object-orientation to a traditional programming language and development environment. Bjarne Stroustrup should be recognized for having the insight to put the two technologies together; he ventured into OO not only before solutions were known to many issues, but before the issues were even widely recognized. He deserves better than a back full of arrows. But in retrospect, we now treat concepts such as multiple inheritance with a good deal of respect, and realize that the Unix development environment with limited linker support does not provide enough compiler support for many of the features that should be in a high level language.(C++ was developed in the mid-eighties supported non-specialized tools found in the Unix program development environment.)

There are solutions to the problems that C++ uncovered. C++ has gone down a path in research, but now we know what the problems are and how to solve them. Let's adopt or develop such languages.

Fortunately, such languages have been developed, which are of industrial strength, meant for commercial projects, and are not just academic research projects. It is now up to the industry to adopt them on a wider scale.

C++, however, retains the problems of the old order of software production. C++ has an advantage over C as it supports many facets of object-orientation. These can be used for some analysis and design. The processes of analysis, design, and organization, however, are still largely external to C++. C++ has not realized the important advantages of integrated software development that leads to improved economies of software production.

Java is an interesting development taking a different approach to C++: s*trict compatibility with C is not seen as a relevant goal* contrary to the initial goal of C++ to be as compatible as possible with C. Java is not the only C based alternative to C++ in the object-oriented world. There has also been Objective-C from Brad Cox, and mainly used in NeXT's OpenStep environment. Objective-C is more like Smalltalk, in that all binding is done dynamically at run time.

A language should not only be evaluated from a technical point of view, c**onsidering its syntactic and semantic features**; it should also be analyzed from the viewpoint of its contribution to the entire software development process. A language should **enable communication between project members** acting at different levels, from management, who set enterprise level policies, to testers, who must test the result. All these people are involved in the general activity of programming, so a language should enable communication between project members separated in space and time. A single programmer is not often responsible

for a task over its entire lifetime, let alone responsible for the whole development process and product.

2.1 Programming

Programming and specification are now seen as the same task. One man's specification is another's program. Eventually you get to the point of processing a specification with a compiler, which generates a program which actually runs on a computer. Carroll Morgan banishes the distinction between specifications and programs: "To us they are **all** programs." [Morgan 90]. Programming is a term that not only refers to implementation; programming refers to the whole process of analysis, design and implementation, deployment and maintenance during its life-time until its retirement.

The Eiffel language integrates the concept of specification and programming, rejecting the divided models of the past in favor of a new integrated approach to projects. Eiffel achieves this in several ways:

it has a clean clear syntax which is easy to read, even by non-programmers;

it has techniques such as preconditions and postconditions so that the semantics of a routine can be clearly documented, these being borrowed from formal specification techniques, but made easy for the 'rest of us' to use;

and it has tools to extract the abstract specification from the implementation details of a program.

Thus Eiffel is more than just a language, providing a whole integrated development environment.

Chris Reade [Reade 89] gives the following explanation of programming and languages.

> *"One, rather narrow, view is that a **program** is a sequence of instructions for a machine. We hope to show that there is much to be gained from taking the much broader view that programs are descriptions of values, properties, methods, problems and solutions. The role of the machine is to speed up the manipulation of these descriptions to provide solutions to particular problems. A **programming language** is a convention for writing descriptions which can be evaluated."*

[Reade 89] also describes programming as being a "*Separation of concerns*". He says:

"The programmer is having to do several things at the same time, namely,

(1) describe what is to be computed;

(2) organize the computation sequencing into small steps;

(3) organize memory management during the computation."

Reade continues, "*Ideally, the programmer should be able to concentrate on the first of the three tasks (describing what is to be computed) without being distracted by the other two, more administrative, tasks. Clearly, administration is important but by separating it from the main task we are likely to get more reliable results and we can ease the programming problem by automating much of the administration.*

The separation of concerns has other advantages as well. For example, program proving becomes much more feasible when details of sequencing and memory management are absent from the program. Furthermore, descriptions of what is to be computed should be free of such detailed step-by-step descriptions of how to do it if they are to be evaluated with different machine architectures. Sequences of small changes to a data object held in a store may be an inappropriate description of how to compute something when a highly parallel machine is being used with thousands of processors distributed throughout the machine and local rather than global storage facilities.

Automating the administrative aspects means that the language implementor has to deal with them, but he/she has far more opportunity to make use of very different computation mechanisms with different machine architectures."

These quotes from Reade are a good summary of the principles from which I criticize C++. What Reade calls administrative tasks, I call *bookkeeping*. Bookkeeping adds to the cost of software production, and reduces flexibility which in turn adds more to the cost. C and C++ are often criticized for being cryptic. The reason is that C concentrates on points 2 and 3, while the description of what is to be computed is obscured.

High level languages describe 'what' is to be computed; that is the problem domain. 'How' a computation is achieved is in the low-level machine-oriented deployment domain. Automating the bookkeeping tasks enhances correctness, compatibility, portability and efficiency. Bookkeeping tasks arise from having to specify 'how' a computation is done. Specifying 'how' things are done in one environment hinders portability to other platforms. The most significant way high level languages replace bookkeeping is using a

declarative approach, (functional as well as logical programming languages.) whereas low level languages use operators, which make them more like assemblers. C and C++ provide operators rather than the declarative approach, so are low level. The declarative approach centralizes decisions and lets the compiler generate the underlying machine operators. With the operator approach, the bookkeeping is on the programmer to use the correct operator to access an entity, and if a decision changes, the programmer will have to change all operators, rather than change the single declaration and simply recompiling. Thus in C and C++ the programmer is often concerned with the access mechanisms to data, whereas high level program development and maintenance far more flexible.

While C and C++ syntax is similar to high level language syntax, C and C++ cannot be considered high level, as they do not remove bookkeeping from the programmer that high level languages should, requiring the compiler to take care of these details. The low level nature of C and C++ severely impacts the development process. The most important quality of a high level language is to remove bookkeeping burden from the programmer in order to enhance speed of development, maintainability and flexibility. This attribute is more important than object-orientation itself, and should be intrinsic to any modern programming paradigm. C++ more than cancels the benefits of OO by requiring programmers to perform much of the bookkeeping instead of it being automated.

Specific examples of bookkeeping: where to allocate memory to an entity (registers, stack, heap, specific global region); memory allocation management. Specific methods for memory allocation; decisions about functions being inline, low level methods for data sharing among functions, modules, actual files or other program components; checks on legality of current value contained by a variable which may contain values outside range expected by programmer; syntactic difference in different constructs which may have same or similar semantics, but the differences are required by the language.

Languages which force a programmer to deal with these issues has broken the separation of concerns. A programmer should only specifies what needs to be computed at the most abstract possible way; supporting tools will target the specification to a concrete implementation.

The industry should be moving towards these ideals, which will help in the economic production of software, rather than the costly techniques of today (this paper was written starting in '90). We should consider what we need, and assess the problems of what we have against that. Object-orientation provides one solution to these problems. The effectiveness of OO, however, depends on the quality of its implementation.

2.2 Communication, abstraction and precision

The primary purpose of any language is communication. A specification is communication from one person to another entity of a task to be fulfilled. At the lowest level, the task to be fulfilled is the execution of a program by a computer. At the next level it is the compilation of a program by a compiler. At higher levels, specifications communicate to other people what is to be accomplished by the programming task. At the lowest level, instructions must be precisely executed, but there is no understanding; it is purely mechanical. At higher levels, understanding is important, as human intelligence is involved, which is why enlightened management practices emphasize training rather than forced processes. This is not to say that precision is not important; precision at the higher levels is of utmost importance, or the rest of the endeavor will fail. Most projects fail due to lack of precision in the requirements and other early stages. Unfortunately, often *those who are least skilled in programming work at the higher levels*, so specifications lack the desirable properties of abstraction and precision. Just as in the *Dilbert Principle* [Adams 96], *the least effective programmers are promoted to where they will seemingly do the least damage*. This is not quite the winning strategy that it seems, as that is where they actually do the most damage, as teams of confused programmers are then left to straighten out their specifications, while the so called analysts move onto the next project or company to sew the seeds of disaster there. (Indeed, since many managers have not read or understood the works of Deming [Deming 82], [L&S 95], De Marco and Lister [DM&L 87], and Tom Peters' later works, the message that the physical environment and attitudes of the work place leads to quality has not got through. Perhaps the humor of Scott Adams is now the only way this message will have impact.)

At higher levels, abstraction facilitates understanding. Abstraction and precision are both important quali-

ties of high level specifications. Abstraction does not mean vagueness, nor the abandonment of precision. Abstraction means the removal of irrelevant detail from a certain viewpoint. With an abstract specification, you are left with a precise specification; precisely the properties of the system that are relevant. Abstraction is a fundamental concept in computing. Aho and Ullman say "An important part of the field [computer science] deals with how to make programming easier and software more reliable. But fundamentally, computer science is a science of *abstraction* -- creating the right model for a problem and devising the appropriate mechanizable techniques to solve it." [Aho 92].

They also say "Abstraction in the sense we use it often implies simplification, the replacement of a complex and detailed real-world situation by an understandable model within which we can solve the problem."

A well known example that exhibits both abstraction and precision is the London Underground map designed by Harold Beck. This is a diagrammatic map that has abstracted irrelevant details from the real London geography to result in a conveniently sized and more readable map. Yet the map precisely shows the underground stations and where passengers can change trains. Many other city transport systems have adopted the principles of Beck's map. Using this model passengers can easily solve such problems as "How do I get from Knightsbridge to Baker Street?"

2.3 Notation

A programming language should support the ex-change of ideas, intentions, and decisions between project members; it should provide a formal, yet readable, notation to support consistent descriptions of systems that satisfy the requirements of diverse problems. A language should also provide methods for automated project tracking. This ensures that modules (classes and functionality) that satisfy project requirements are completed in a timely and economic fashion. A programming language aids reasoning about the design, implementation, extension, correction, and optimization of a system.

During requirements analysis and design phases, formal and semi-formal notations are desirable. Notations used in analysis, design, and implementation phases should be complementary, rather than contradictory. Currently, analysis, design and modeling notations are too far removed from implementation, while programming languages are in general *too low level*. Both designers and programmers must compromise to fill the gap.

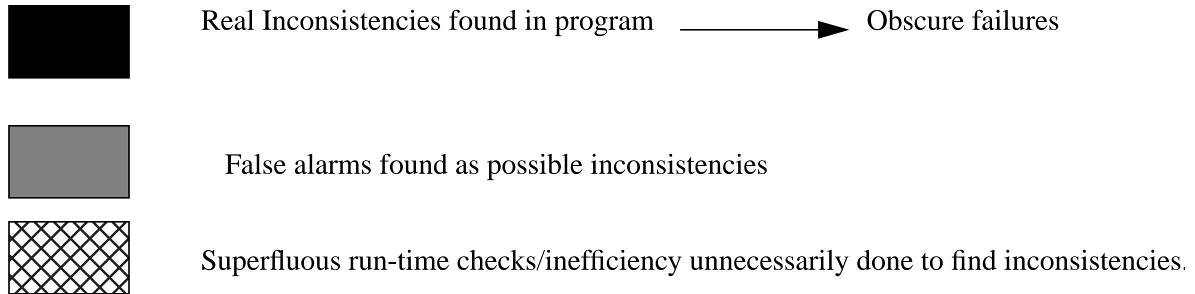
Many current notations provide difficult transition paths between stages. This 'semantic gap' contributes to errors and omissions between the requirements, design and implementation phases. Better programming languages are an implementation extension of the high level notations used for requirements analysis and design, which will lead to improved consistency between analysis, design and implementation. Object-oriented techniques emphasize the importance of this, as abstract definition and concrete implementation can be separate, yet provided in the same notation. Programming languages also provide notations to formally document a system. **Program source is the only reliable documentation of a system**, so a language should explicitly support documentation, not just in the form of comments. As with all language, the effectiveness of communication is dependent upon the skill of the writer. Good program writers require languages that support the role of documentation, and that the language notation is perspicuous, (free from obscurity or ambiguity) and easy to learn. Those not trained in the skill of 'writing' programs, can read them to gain understanding of the system. After all, it is not necessary for newspaper readers to be journalists.

2.4 Tool Integration

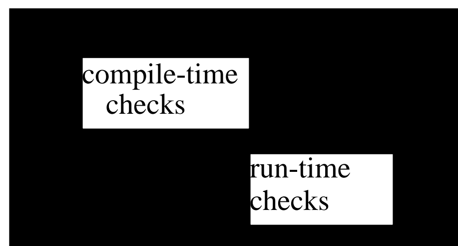
A language definition should enable the development of integrated automated tools to support software development. For example, browsers, editors and debuggers. The compiler is just another tool, having a twofold role. Firstly, code generation for the target machine. The role of the machine is to execute the produced programs. A compiler has to check that a program conforms to the language syntax and grammar, so it can 'understand' the program in order to translate it into an executable form. Secondly, and more importantly, the compiler should check that the programmers expression of the system is valid, complete, and consistent; i.e., perform semantics checks that a program is internally consistent. Generating a system that has detectable inconsistencies is pointless.

2.5 Correctness

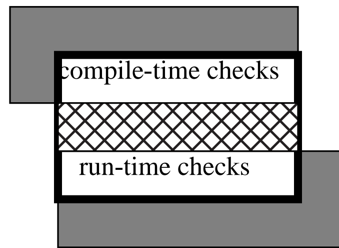
Deciding what constitutes an inconsistency and how to detect it often raises passionate debate. The discord arises because the detectable inconsistencies do not exactly match real inconsistencies. There are two opposing views: firstly, languages that overcompensate are restrictive, you should trust your programmers; secondly, that programmers are human and make mistakes and program crashes at run-time are intolerable. This is the key to the following diagrams: we are looking to write programs and tools that support that development such that we end with a program with no inconsistencies.



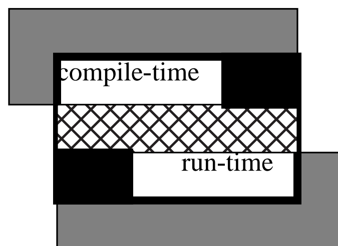
In the first figure the black box represents the real inconsistencies, which must be covered by either compile-time checks or run-time checks; that is, those two should help their detection.



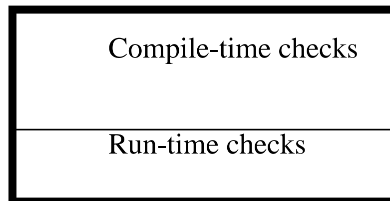
In the scenario of this diagram, checks are insufficient so obscure failures occur at run-time, varying from obscure run-time crashes to strangely wrong results to being lucky and getting away with it. Currently too much software development is based on programming until you are in the lucky state, known as *hacking*. This sorry situation in the industry must change by the adoption of better languages to remove the *ad hoc* nature of development.
Some feel that compiler checks are restrictive and that run-time checks are not efficient, so passionately defend this model, as programmers are supposedly trustworthy enough to remove the rest of the real inconsistencies. Although most programmers are conscientious and trustworthy people, this leaves too much to chance. You can produce defect-free software this way, as long as the programmer does not introduce the inconsistencies in the first place, but this becomes much more difficult as the size and complexity of a software system increases, and many programmers become involved. The real inconsistencies are often removed by hacking until the program works, with a resultant dependency on testing to find the errors in the first place. Sometimes companies depend on the customers to actually do the testing and provide feedback about the problems. While fault reporting is an essential path of communication from the customer, it must be regarded as the last and most costly line of defence. C and C++ are in this category. Software produced in these languages is prone to obscure failures.



The second figure, shows that the language detects inconsistencies beyond the real inconsistency box. These are false alarms. The run-time environment also doubles up on inconsistencies that the compiler has detected and removed, which results in run-time inefficiency. The language will be seen as restrictive, and the run-time as inefficient. You won't get any obscure crashes, but the language will get in the way of some useful computations. Pascal is often (somewhat unfairly) criticized for being too restrictive.
The figure below shows an even worse situation, where the compiler generates false alarms on fictional inconsistencies, does superfluous checks at run-time, but fails to detect real inconsistencies.



The best situation would be for a compiler to statically detect all inconsistencies without false alarms. However, it is not possible to statically detect all errors with the current state of technology, as a significant class of inconsistencies can only be detected at run-time; inconsistencies such as: divide by zero; array index out of bounds; and a class of type checks that arise during run-time.



The current ideal is to have the detectable and real inconsistency domains exactly coincide, with as few checks left to run-time as possible. This has two advantages: firstly, that your run-time environment will be a lot more likely to work without exceptions, so your software is safer; and secondly, that your software is more efficient, as you don't need so many run-time checks. A good language will correctly classify inconsistencies that can be detected at compile time, and those that must be left until run-time.
This analysis shows that as some inconsistencies can only be detected at run-time, and that such detection results in exceptions that exception handling is an exceedingly important part of software. Unfortunately, exception handling has not received serious enough attention in most programming languages.
Eiffel has been chosen for comparison in this critique as the language that is as close to the ideal as possible; that is, all inconsistencies are covered, while false alarms are minimized, and the detectable inconsistencies are correctly categorized as compile-time or run-time. Eiffel also pays serious attention to exception handling.

2.6 Types

In order to produce correct programs, syntax checks for conformance to a language grammar are not sufficient: we should also check semantics. Some semantics can be built into the language, but mostly this must be specified by the programmer about the system being developed (via user-defined types).
Semantics checking is done by ensuring that a specification conforms to some schema. For example, the sentence: "The boy drank the computer and switched on the glass of water" is grammatically correct, but nonsense: it does not conform to the mental schema we have of *computers* and *glasses of water.* A programming language should include techniques for the detection of similar nonsense. The technique that enables detection of the above nonsense is types. We know from the computer's *type* that it does not have the property 'drinkable'. Types define an entity's properties and behavior.
Programming languages can either be typed or untyped; typed languages can be statically typed or dynamically typed. Static typing ensures at compile time that only valid operations are applied to an entity. In dynamically typed languages, type inconsistencies are not detected until run-time. Smalltalk is a dynamically typed language, not an untyped language. Eiffel is statically typed. C++ is statically typed, but there are many mechanisms that allow the programmer to render it effectively untyped, which means errors are not detected until a serious failure. Some argue that sometimes you might want to force someone to drink a computer, so without these facilities, the language is not flexible enough. The correct solution though is to modify the design, so that now the computer has the property drinkable. Undermining the type system is not needed, as the type system is where the flexibility should be, not in the ability to undermine the type system. **Providing and modifying declarations** is declarative programming.
Eiffel tends to be declarative with a simple operational syntax, whereas C++ provides a plethora of operators. Defining complex types is a central concept of object-oriented programming: "Perhaps the most important development [in programming languages] has been the introduction of features that support abstract data types (ADTs). These features allow programmers to add new types to languages that can be treated as though they were primitive types of the language. The programmer can define a type and a collection of constants, functions, and procedures on the type, while prohibiting any program using this type from gaining access to the implementation of the type. In particular, access to values of the type is available only through the provided constants, functions, and procedures." [Bruce 96].
Object-oriented programming also provides two specific ways to assemble new and complex types: "objects can be combined with other types in expressive and efficient ways (composition and hierarchy) to define new, more complex types." [Ege 96].

2.7 Redundancy and Checking

Redundant information is often needed to enable correctness checking. Type definitions define the elements in a system's universe, and the properties governing the valid combinations and interactions of the elements. Declarations define the entities in a system's universe. The compiler uses redundant information for consistency checking, and strips it away to produce efficient executable systems. Types are redundant information. You can program in an entirely typeless language: however, this would be to deny the progress that has been made in making programming a disciplined craft, that produces correct programs economically.
It is a misconception that consistency checks are 't*raining wheels*' for student programmers, and that 'syntax' errors are a hindrance to professional programmers. Languages that exploit techniques of schema checking are often criticized as being restrictive and therefore unusable for real world software. This is nonsense and misunderstands the power of these languages. It is an immature conception; the best programmers realize that programming is difficult. As a whole, the computing profession is still learning to program.
While C++ is a step in this direction, it is hindered by its C base, importing such mechanisms as pointers with which you can undermine the logic of the type system. Java has abandoned these C mechanisms where they hinder: "The Java compiler employs stringent compile-time checking so that syntax-related errors can be detected early, before a program is deployed in service" [Sun 95]. The programming community has matured in the last few years (or so we hope), and while there was vehement argument against such checking in the past by those who saw it as restrictive and disciplinarian, the majority of the industry now accepts, and even demands it.
Checking has also been criticized from another point of view. This point of view says that checking cannot

guarantee software quality, so why bother? The premise is correct, but the conclusion is wrong. Checking is neither necessary, nor sufficient to produce quality software. However, it is helpful and useful, and is a piece in a complicated jig-saw which should not be ignored. In fact there are few things that are necessary for quality software production. Mainly, software quality is dependent on the skill and dedication of the people involved, not methodologies or techniques. There is nothing that is sufficient. As Fred Brooks has pointed out, there is no *Silver Bullet* [Brooks 95]. Good craftsmen choose the right tools and techniques, but the result is dependent on the skill used in applying the tools. Any tool is worthless in itself. But the *Silver Bullet* rationale is not a valid rationale against adopting better programming languages, tools and environments; unfortunately, Brooks' article has been misused.

Another example of consistency checking comes from the user interface world. Instead of correcting a user after an erroneous action, a good user interface will not offer the action as a possibility in the first place. It is cheaper to avoid error than to fix it. Most people drive their cars with this principle in mind: smash repair is time consuming and expensive. Program development is a dynamic process; program descriptions are constantly modified during development. Modifications often lead to inconsistencies and error. Consistency checks help prevent such 'bugs', which can 'creep' into a previously working system. These checks help verify that as a program is modified, previous decisions and work are not invalidated.

It is interesting to consider how much checking could be integrated in an editor. The focus of many current generation editors is text. What happens if we change this focus from text to program components? Such editors might check not only syntax, but semantics. Signalling potential errors earlier and interactively will shorten development times, alerting programmers to problems, rather than wasting hours on changes which later have to be undone. Future languages should be defined very cleanly in order to enable such editor technology.

2.8 Encapsulation

There is much confusion about encapsulation, mostly arising from C++ equating encapsulation with *data hiding*. The Macquarie dictionary defines the verb *to encapsulate* as "*to enclose in or as in a capsule.*" The object-oriented meaning of encapsulation is to enclose related data, routines and definitions in a class capsule. This does not necessarily mean hiding.

Implementation hiding is an orthogonal concept which is possible because of encapsulation. Both data and routines in a class are classified according to their role in the class as interface or implementation. To put this another way: first you encapsulate information and operations together in a class, then you decide what is visible, and what is hidden because it is implementation detail. Most often only the interface routines and data should appear at design time, the implementation details appearing later.

Encapsulation provides the means to separate the abstract interface of a class from its implementation: the interface is the visible surface of the capsule; the implementation is hidden in the capsule. The interface describes the essential characteristics of objects of the class which are visible to the exterior world. Like routines, data in a class can also be divided into characteristic interface data which should be visible, and implementation data which should be hidden. Interface data are any characteristics which might be of interest to the outside world. For example when buying a car, the purchaser might want to know data such as the engine capacity and horse-power, etc. However, the fact that it took John Engineer six days to design the engine block is of no interest. Implementation hiding means that data can only be manipulated, that is updated, within the class, but it does not mean hiding interface data. If the data were hidden, you could never read it, in which case, classes would perform no useful function as you could only put data into them, but never get information out. In order to provide implementation hiding in C++ you should access your data through C functions. This is known as data hiding in C++. It is not the data that is actually being hidden, but the access mechanism to the data. The access mechanism is the implementation detail that you are hiding. C++ has visible differences between the access mechanisms of constants, variables and functions. There is even a typographic convention of upper case constant names, which makes the differences between constants and variables visible.Java has these idiosyncrasies as well.

The fact that an item is implemented as a constant should also be hidden. Most non-C languages provide uniform functional access to constants, variables and value returning routines. In the case of variables,

functional access means they can be read from the outside, but not updated. An important principle is that updates are centralized within the class.

Above I indicated that encapsulation was grouping operations and information together. Where do functions fit into this? The wrong answer is that functions are operations. Functions are actually part of the information, as a function returns information derived from an object's data to the outside world. This theme and its adverse consequences, that place the burden of encapsulation on the programmer rather than being transparent, recur throughout this critique.

2.9 Safety and Courtesy Concerns

This critique makes two general types of criticism about 'safety' concerns and 'courtesy' concerns. These themes recur throughout this critique, as C and C++ have flaws that often compromise them. Safety concerns affect the *external* perception of the quality of the program; failure to meet them results in unfulfilled requirements, unsatisfied customers and program failures.

Courtesy concerns affect the *internal* view of the quality of a program in the development and maintenance process. Courtesy concerns are usually stylistic and syntactic, whereas safety concerns are semantic. The two often go together. It is a courtesy concern for an airline to keep its fleet clean and well maintained, which is also very much a safety concern.

Courtesy issues are even more important in the context of reusable software. Reusability depends on the clear communication of the purpose of a module. Courtesy is important to establish social interactions, such as communication. Courtesy implies inconvenience to the provider, but provides convenience to others. Courtesy issues include choosing meaningful identifiers, consistent layout and typography, meaningful and non-redundant commentary, etc. Courtesy issues are more than just a style consideration: a language design should directly support courtesy issues. A language, however, cannot enforce courtesy issues, and it is often pointed out that poor, discourteous programs can be written in any language. But this is no reason for being careless about the languages that we develop and choose for software development.

Programmers fulfilling courtesy and safety concerns provide a high quality service fulfilling their obligations by providing benefits to other programmers who must read, reuse and maintain the code; and by producing programs that delight the end-user.

The *programming by contract* model has been advocated in the last few years as a model for programming by which safety and courtesy concerns can be formally documented. Programming by contract documents the obligations of a client and the benefits to a provider in preconditions; and the benefits to the client and obligations of the provider in postconditions [Meyer 88], [Kilov and Ross 94].

2.10 Implementation and Deployment Concerns

Class implementors are concerned with the implementation of the class. Clients of the class only need to know as much information about the class as is documented in the abstract interface. The implementation is otherwise hidden. Another aspect that is just as important to shield programmers from is deployment concerns.

Deployment is how a system is installed on the underlying technology. If deployment issues are built into a program, then the program lacks portability, and flexibility. One kind of deployment concern is how a system is mapped to the available computing resources. For example, in a distributed system, this is what parts of the system are run in which location. As things can move around a distributed system, programmers should not build into their code location knowledge of other entities. Locations should be looked up in a directory. Another deployment issue is how individual units of a system are plugged together to form an integrated whole. This is particularly important in OO, where several libraries can come from different vendors, but their combination results in conflicts. A solution to this is some kind of language that binds the units. Thus if you purchase two OO libraries, and they have clashes of any kind, you can resolve this deployment issue without having to change the libraries, which you might not be able to do anyway. Programmers should not only be separated from implementation concerns of other units, but separated from deployment concerns as well.

2.11 Concluding Remarks

It is relevant to ask if grafting OO concepts onto a conventional language (like Objective-C onto C, C++

onto C, Ada-95 onto Ada-83, and to a large extend like Java onto C) releases the full benefits of OO? The following parable seems apt: “No one sews a patch of unshrunk cloth on to an old garment; if he does, the patch tears away from it, the new from the old, and leaves a bigger hole. No one puts new wine into old wineskins; if he does, the wine will burst the skins, and then wine and skins are both lost. New wine goes into fresh skins.” *Mark 2:22*

We must abandon disorganized and error-prone practices, not adapt them to new contexts. How well can hybrid languages support the sophisticated requirements of modern software production? In my experience *bolt-on* approaches to object-orientation usually end in disaster, with the new tearing away from the old leaving a bigger hole.

Surely a basic premise of object-oriented programming is to enable the development of sophisticated systems through the adoption of the simplest techniques possible? Software development technologies and methodologies should not impede the production of such sophisticated systems.