Objectware : from C++ towards CAD++

Zenon Rychter
Technical University of Bialystok
Poland

Objectware is software supporting the object-oriented paradigm. Object orientation controls complexity through thinking in natural terms. The approach unifies all stages of complex system development: analysis, design, and programming. It applies to software design, from operating systems to CAD packages, as well as all fields of engineering design, CAD based or not. The paper discusses what next generation CAD, object-oriented CAD or CAD++ will be like by studying the philosophy behind the C++ object-oriented programming language, which most CAD++ software developers use.

Objectware: from C++ towards CAD++

Object revolution

We are witnessing a software revolution. The name is object-orientation. Software turns to objectware. Object-oriented programming languages are mushrooming, see Coad(1991a, 1991b, 1993), Goldberg(1985), Jamsa(1996), Keene(1989), Perry(1994), Stroustrup(1991). Good old procedural languages convert to object-orientation. The C language has grown into C++, the two plusses meaning the added value of being object-oriented. Along a parallel path C has become Objective-C. The Pascal language has evolved into Object Pascal. Even Basic has acquired some object feel. And those not lured so far perhaps will find more visual appeal in: Visual C++ or Visual Basic. Languages such as Smalltalk and Eiffel are object-oriented from the outset, with no pre-object-oriented parents.

Language, human or programming, is important: it decides what and how we can think. It provides building blocks and construction rules for modeling reality. Equally important, it enables communication: human to human, application to application, human to application, and application to human. Finally, it provides long-term storage of knowledge. But unhandy language obstructs construction or even makes it impossible. And communication may stuck in a bottleneck, suffer from noise, or even break down.

The past: from machine code to C

The evolution of human and programming languages followed different paths, sometimes running in opposite directions. Humans initially used pictorial hieroglyphics, which represented things in a visual, direct, self-explanatory way. In computer parlance, it was an iconic, WYSIWYG, What You See Is What You Get, approach, the ultimate stage of modern Graphical User Interfaces (GUIs) and visual programming. Hieroglyphs (icons) are visually attractive to the novice, but are troublesome in more demanding tasks. Since each icon is (supposed to be, not necessarily in reality) meaningful, a new icon must be designed (and there are no standards here) whenever a new name is needed. In a rapidly growing world, this must end in an iconic explosion. A meaningful, individual icon cannot be used as an anonymous, standard, universal, atomic building block in larger, hierarchical constructs. Its considerable size leads to storage overload when it comes to huge collections of data. Humanity (with the notable exceptions of China, Japan, and Korea) solved those problems by the introduction of small, fixed, meaningless, abstract alphabets. So characters replaced hieroglyphs in human languages. By contrast, character-mode interaction with computers (through keyboards and typewriter-like printers) was the original approach, now generally regarded obsolete, not user-friendly.

The earliest programming languages were totally unlike the primitive human languages. They were direct manipulators of computer memory, organized in an array of two-state (zero-one = bit) switching devices. Quite rightly their name was machine languages, or rather codes, something not to be read by humans. The gap between end users (speaking plain prose) and programmers (bit shifters) was at its widest.

To ease the pain of programming in a machine language , assembler languages were introduced, see Duntemann(1993). Symbolic names such INC (for increase) or JPM (for jump) could now be used by programmers to replace sheer numbers understood by computers. To end users the advance was microscopic. The tool was not for them at all.

The introduction of the FORTRAN (formula translator) language initiated the epoch of procedural programming. Now actions could be represented by functions. But the functions operated on rather raw data: arrays of numbers were the limit of sophistication. Number-crunching has become easy. Those interested in conceptual, creative models of complex realities had still to wait for something else.

Necessity is the mother of invention. Computers themselves became complex systems involving many different components (processors, random access memory, permanent storage, input devices, displays, printers) communicating with each other. Modeling (thinking about) computers in terms of numbers just was no longer feasible. System programming was required and the C language was developed as the standard for writing operating systems (OS), see Kernighan(1988). C is a small, portable (machine independent) general purpose language. With C you have almost absolute control over the computer, down to the granularity of an individual bit of memory. At the same time, you can form hierarchical constructs of any complexity, representing any conceivable data structure from any application domain: numbers, arrays, arrays of arrays (of arrays, etc. if necessary), vectors, lists, queues, graphs, trees, records, files, sets, enumerations, characters, strings, sentences, paragraphs, books, dictionaries, geometrical entities. In short: objects. Real life objects. Objects for all. No wonder the C language has become the work-horse tool of professional software developers in all areas: compilers of programming languages, graphical user interfaces (GUIs), text processors, data bases, spreadsheets, calculators, symbol manipulators, simulators, games, and computer aided design (CAD) systems. Third-party developers of add-on extensions to CAD packages also resorted to C when performance and seamless integration with the core CAD system were at stake. There is no secret to that: C is the true engine of CAD and an open CAD's application programming interface (API) is most naturally a version of C, perhaps wrapped up in domain specific naming. So whoever you are, and whatever you need to model, C can do the job.

The present: from C to C++

Yet people solving real life problems, including designers using CAD, have not turned to C as a general conceptual framework, thinking tool,

and communication standard. Even C programmers were looking for something that would make their work easier. Life with C was exciting but risky and error prone: full control and full responsibility for what you are doing. This was good for one person, solving a simple problem in a short time. It was painfully inadequate in a complex task, requiring the concerted effort of many people with different fields of expertise, spanning a long period of time, going iteratively and backtracking through phases as diverse as making an initial rough concept and implementing it in full detail, prototyping, testing, fine tuning. Here complexity and change rule. Reliance on a human coordinator who understands all, remembers everything, keeps track of all changes and sees all their effects and side effects would lead to disaster. The answer is organization enforced by a language. The language would have to formal because computers do not accept informal partners. On the other hand, the language should be natural and flexible enough to reflect the human way of thinking, so that everybody involved in a complex project could use it.

And so the C++ language was born, an object-oriented extension of C. It is discussed here because it is the most widely used object-oriented language. Importantly, the author uses it. And the design principles behind C++ are profoundly expounded by the language originator, Stroustrup(1991). Finally, the transition from C to C++ suggests the name CAD++ for an object-oriented, next generation CAD. But what counts here is not one language or another, but the underlying object oriented philosophy, shared by all languages of the class. A philosophy of interest to all: CAD software developers, third-party developers, and common designers using CAD, see Grabowski(1995).

While C++ expands C and accepts all of the C-language constructs, the changes it brings are revolutionary. It is not that an application developed using C++ will necessarily differ from one developed by means of C. It is the whole uphill way from an initial vague idea of an application to its final production version that has been dramatically changed. C++ addresses several fundamental issues: complexity, reuse (multiple use), expansion, maintenance, flexibility, error sensitivity reduction, change sensitivity localization, concurrent, distributed development at different levels, meaningful and natural cross-level communication, access control, the economy of the whole software development process.

On the surface, the main thing is an object, as the name object-orientation suggests. Objects represent non-volatile granules from the world around us. Object should be as natural and obvious as possible, and should be given self-explanatory names (e.g. a room). And what is natural is decided by the end user, CAD user, not the programmer. The end user selects objects of interest in his domain and he names them as he likes. Hence object-orientation is end-user orientation. That's a revolution. It not only pleases end users, but promotes stability, since domain specific knowledge will probably change little compared to changes in information technology.

Useful objects come in packs, types, where otherwise different individuals have something important in common. In C++ types are

called classes, and thinking in terms of classes is the true solid foundation of the object-oriented approach. A class is a type (or prototype), a template for object (real thing) construction, destruction, and behavior. A class is a fully automated object factory, with customized output. For a one-time design task a hand-made object will suffice. But you will not reuse it: not a car but the car factory is reusable if you need another car. A class abstracts structure from objects. An object is an instantiation of its class. Abstraction takes extra effort, and deeper knowledge, but pays off abundantly in the long run. A well designed and carefully tested class supports reuse. You do not have to start from scratch but construct from safe, standard, customizable components.

A class is a self-contained, all-in-one module. It encapsulates data (attributes, properties) and functions operating on the data. This makes for intelligent objects, objects that "know" how to behave depending on circumstances. For example, a room object knows that it cannot overlap other rooms and that its proportions must respect practical limits.

As much as possible, a class is a black box. A good example is a TV set. It is operated by a simple user interface (GUI in fact), but the details of construction (termed implementation in programming parlance) are hidden. It only takes one push of a button to turn your TV on and see its default behavior. Whenever needed, the behavior can be adjusted. When you create on object of a class, you turn it on. Objects also have default behavior, so you do not have to know much to let objects fly. If you learn about them more, your control will increase.

The distinction between class interface and implementation is of fundamental importance for all: CAD systems manufacturers, third-party developers of add-ons, power users, and ordinary end users. The ordinary end user will only have to do with what is called public interface. Functions that compose this interface are just object manipulators for everyday use. For example, windows, dialog boxes, menus, toolboxes are all objects operated via public, graphical interface. The public interface of a well designed class is the most durable part of a system. A public function *rotate(angle, angle, angle)* rotating an object in three dimensions about three axes will never change, because its logic is fixed. In C++ you may use functions with the same name for rotations about two or one axis, by just writing *rotate(angle, angle)* or *rotate(angle)*, respectively. Even more, if you want to incrementally rotate by a fixed angle about a fixed axis, you can just write *rotate()*, and the system will know what to do. So functions in C++ are also intelligent: under one name you group akin but not identical operations, and depending on circumstances (e.g. number of supplied angles of rotation) the system will call the right version of operation. In human communication context-sensitive naming is commonplace. Now programming languages are following suit.

Users operate objects via public interface functions, which, by design, are as close as possible to the user natural mode of communication. When existing objects need to be refined, or new objects are required, it is the end user who starts the process by specifying her/his needs in terms of public interface: functions, their names, their arguments, the values they

return (if any). This wish-list is then transferred for implementation to developers literate in programming, but the process of implementation will not change the interface functions look and behavior. So end users and programmers speak, where feasible, the same language, the end user language. Communication is direct, no technical lingo breaks it. Everybody can concentrate on his job. Programmers may continually improve on implementation according to advances in computer technology (new processors, parallel processing, etc.) without disturbing end users living a stable life among their public interface functions.

A class constructs similar objects, carrying the same level of detail. If there is not enough information to construct objects of a class, the class is called virtual. For example, a class *polygon* may have *area()* among its public functions, but it may not be clear how to calculate the area of a general polygon. Still a lot of useful information can be placed in that class (vertices, edges), and only the area calculation must be postponed until later. A virtual class is always a base class or parent for derived or child classes. A child inherits everything from its parent, defines the virtual (undefined) parts of its parent, and adds extra data and functionality. For example, *rectangle* may be derived from *polygon*. The *area*() function for a rectangle can be easily defined. *Rectangle* can also have a *diagonal()* public function, not present in the base class *polygon*, where it does not make sense. A child can be a parent of another class, and so on. A child can have, one, two, or more parents, and inherit from all of them (multiple inheritance). So classes provide a powerful tool for the modeling of hierarchical relationships, having the form of directed acyclic graphs or trees. Cyclic relationships are also accounted for through friend classes. A friend of a class must be declared within the class and has full access to all of its contents.

The concept of base and derived classes is fundamental to class reuse, expansion, and maintenance. You do not have to start from scratch, but search an available class library for closest match to your needs. For example, if you have a *rectangle* class and want to develop a *room* class, you may derive *room* from *rectangle* and inherit all geometrical data and functionality ready-made in *rectangle*. You only have to add what is specific to *room* and not present in *rectangle*. This approach saves a lot of effort. The future of CAD will definitely look that way. CAD software manufacturers provide the core CAD engine and base classes. Third-party developers derive from base classes and supply specialized child class libraries. Power users customize and expand functionality using class-based APIs. Ordinary end users just use objects in their daily work, but when not satisfied they send complaints and wish-lists, specifying expected changes to public interface functions, secure in the knowledge that in a class-oriented environment their dreams may promptly come true.

While ordinary users will use only the public interface, developers of derived classes (who as everybody else have access to the public part of a class) have also their own, protected interface. The protected components of a class are hidden from the general public. But they are visible and can be used in derived classes.

At the deepest level of protection in a class are its private data and functions. These are the innermost details of class construction, its internal organs. Ordinary mortals would respect privacy and stay away from the private and even protected class members. If they are to change private data, they have to do so through public interface. For example, you change the contrast of your display by turning a knob, not manipulating the internal wiring. This public/protected/private organization keeps everything and everybody in order. Without it everybody could change everything at any moment without the knowledge of her/his partners. In a serious project this would lead to disaster.

In real life we see wholes and their parts. C++ accounts for this through members (parts) of a class (whole). The members can be objects of other classes or they may be classes themselves. The nesting of classes can be as deep as we wish, a framework flexible enough for any problem area. Also recurrent membership is possible, of unlimited depth. So a class can contain objects of the same class, and those objects can also be composed of same class objects, etc. It is like a TV displaying another TV, in which one sees another TV. Fractal images also have such self-similar, self-repeating at different scales structure. In CAD, complex (whole, parts, parts of parts), evolving, branching, recurrent, dynamically growing and shrinking structures are commonplace. C++ handles them all.

A class is a template for objects. But classes themselves may be derived from class templates, providing common functionality for a parametric family of classes with different members having only different types. This is specifically possible for container classes (sets, lists, vectors, queues, dictionaries). C++ provides parametric containers as a library. If you need a class of vectors of a particular type, you take the library template class and simply specify the type of your objects. By a similar mechanism, C++ supports function templates, that is functions which do the same job, sorting for example, on different type objects. The beauty of this is that one has to remember only one name for all possible uses of a function in different contexts.

Objects communicate with each other through the mechanism of messages. The mechanism is particularly important in interactive, multithreaded, multitasking, multi-user environments. For example, a mouse click is turned to a message send to the window on which the mouse cursor was located at the moment of click. The window may handle the message (e.g. repaint itself) or pass it elsewhere. Messages are placed in message queues of particular processes or threads where they wait for handling. Each process has its message loop describing which messages to handle and how. A message-driven environment is ideal for concurrent, distributed CAD.

The future: from CAD to CAD++ or Object Oriented CAD

The future undoubtedly belongs to the object-oriented paradigm. The paradigm really says: think in your professional domain as you think in everyday life. There can be nothing simpler than that. Everybody can

easily convert to this faith. Programmers were the first to do so. Object-oriented programming dominates the 90s. This is the more so that, on top of raw object-oriented programming languages, we have a growing offer of visual programming environments where powerful, full-featured GUI applications are constructed graphically onscreen, by drag-and-drop operations. Windows-based operating systems are not only perfect examples of objects in action, but underneath are complete object-oriented programming environments with ready to use GUI classes. And since far more than 50% of code of an application goes into the GUI department, the increase in speed of software construction is amazing. CAD software developers, where user interaction is the core functionality, are in the same category as GUIs. There is no longer need to reinvent windows, dialog boxes, and the like - those are built-in parts of a windowing system. Even three-dimensional graphics and rendering classes, such as Open GL, are already there. A good taste and feel for the user needs becomes more important in developing a good CAD package than the mysterious skill of coding. So far CAD users and third-party developers could only use objects of predefined classes. They were condemned to the same non-object-oriented C language as the original CAD manufacturer. New generation, object oriented CAD or CAD++ is written in object-oriented C++, so creating new classes (user defined types) becomes an equally easy task for the CAD++ creator, third-party developer of library components, and end user. In such an open object-oriented environment, ready made libraries of all sorts of domain-specific classes will appear at the speed of a chain reaction. And common users will have to do with a myriad of intelligent objects, knowing the rules of geometry, placement, dimensioning, acoustics, lighting, even esthetics, and knowing at what stage of the design process what part of their built-in intelligence to show, from conceptual design, to production drawings, and specification writing. With CAD++ the design process will no longer be as we know it.

References

Coad P., Yourdon E.(1991a), Object-Oriented Analysis, 2nd Ed. Englewood Cliffs, N.J., Prentice Hall.

Coad P., Yourdon E.(1991b), Object-Oriented Design. Englewood Cliffs, N.J., Prentice Hall.

Coad P., Nicola J.(1993), Object-Oriented Programming. Englewood Cliffs, N.J., Prentice Hall.

Duntemann J.(1993), Assembly Language Step-by-Step. New York, N.Y., Wiley.

Goldberg A.(1985), Smalltalk-80: The Interactive Programming Environment. Reading, Mass., Addison-Wesley.

Grabowski R.(1995), Objective MicroStation v6.0. MicroStation User Europe, May/June, 16-17.

Jamsa K.(1996), JAVA Now, Jamsa Press.

Keene S.E.(1989), Object-Oriented Programming in Common LISP. Reading, Mass., Addison-Wesley.

Kernighan B.W., Ritchie D.M.(1988), The C Programming Language. Englewood Cliffs, N.J., Prentice-Hall.

Perry P., Corry C., Cullens C., Davidson M., McKean R.W., Tackett J.(1994), Using Visual C++ 2. Que Corporation.

Stroustrup B.(1991), The C++ Programming Language, 2nd Ed. Reading, Mass., Addison-Wesley.