Of Minimalism, Constructivism and Program Code

In part this essay is a continuation of Kevlin Henney's arguments in "minimalism : omit needless code[1]" but it is also, in part a response and counter argument. Let me say up front: I like minimalism, I like Kevlin's piece and I agree with much of what he says.

However, I worry about some of the advice. I worry about the direction of *modern C++*, I worry about what I perceive to be an increasing elitist attitude in C++ coding. I worry about all these things because of accessibility.

Less is more

First let me take Kevlin's loop example. To remind you, this is a for-loop which prints out the content of a vector, example 1 is Kevlin's first pass using an integer to access the vector.

```
// example 1
std::vector<std::string> items;
...  // populate items
for (std::size_t at = 0; at != items.size(); ++at)
     std::cout << items[at] << std::endl;
```

Next the code is changed to use an iterator:

```
// example 2
std::vector<std::string> items;
... // populate items
for (std::vector<std::string>::iterator at = items.begin();
      at != items.end(); ++at)
      std::cout << *at << std::endl;
```

What has this change given us? In terms of functionality: nothing; in terms of performance, well maybe we save a coupe of index operations; in terms of style it is more "modern" – that is to say it looks more like *modern C++* because we are using iterators.

But in our haste for fashion what have we lost? We have increased code complexity, we are demanding a more detailed knowledge of C++ and its library than we did previously. This isn't a crime, after all, if you don't know iterators how can you claim to know C++?

On balance I think this is a judgement call, it depends on the style of the program. I can't get worked up about this.

After a couple more iterations Kevlin gives us:

```
// example 3
std::vector<std::string> items;
```

```
... // populate items
typedef std::ostream_iterator<std::string> out;
std::copy(items.begin(), items.end(), out(std::cout, "\n");
```

Now we have less code still. However, we have significantly increased the about of context information required to understand the code. We must now understand std::copy, std::ostream_iterator and the magic typedef. Further, we have lost our end-of-line abstraction, std::endl, now we must know the correct end of line terminator – CR? LF? CR and LF? Is this output for the screen where it hardly matters? Or is it to a file where it is more important?

So, although we have less code we actually require more information to understand it. This cuts to the heart of one of the fundamental problems of code reuse: to reuse code, that is, to be able to write less code, we must know more, that is we must increase out knowledge. In this case it is the standard C++ library, yes, every C++ programmer should know the library, but I ask you: do you know the entire library?

Is this really minimalism?

> "The aim of Minimalism is to allow the viewer to experience the work more intensely without the distractions of composition, theme and so on[2]."

Using this definition example 3 is not minimalist because it is full of distractions, to understand the code we must understand the context it is written in – the theme is critical to understanding the code. Example 1 may contain less code but it is actually closer to a minimalist solution.

> "Constructivist art is marked by a commitment to total abstraction and a wholehearted acceptance of modernity.... Objective forms which were thought to have universal meaning were preferred over the subjective or the individual. The art is often very reductive as well, paring the artwork down to its basic elements[3]."

In fact, example 3 is more constructivist in nature than minimalist. It is reduced to the basic elements using every abstraction available. On of the best known constructivists, Wassily Kandinsky created whole pictures from his context theories: "elementary shapes were yellow for the triangle, red for the square and blue for the circle, mixtures of colour would need to follow from mixtures of form. A pentagon mixes a square and triangles: it must therefore be orange[4]."

As in art such minimalism or constructivism can make program code difficult to comprehend.

Need for context

> "For me it was minimalism. I felt at home with it, felt I might of invented it. Yet I have totally failed to write about it..... [the ultra minimalist sculptor] Sandback can transfix and subjugate me with a length of twine strung across a corner of a room but I have found no way to write about the experience[5]."

So wrote the art critic David Sylvester in 1996. Let us examine this for a moment. Here is an acclaimed art critic, a man who is paid to understand and explain modern art, one who we expect to understand Pop Art,

Conceptual Art, Abstract Art and more, yet here he is holding up his hands and saying "I can't explain minimalist art."

As we push further and further onwards in the direction of minimalism it becomes less and less accessible. This is true of art, literature and program code.

Simply claiming that program code is not written by dummies, for dummies is not enough. As professional developers we have a responsibility to ensure our code is maintainable. To paraphrase Arthur C Clark: "any sufficiently advanced software implementation is indistinguishable from unmaintainable code.[6]"

Let us suppose you write you masterpiece of a system and you then leave the project. Who are you going to hand it over too? If you are lucky you have a team of equally capable developers who are ready to take over. More likely as Steve Maguire[7] points it is junior developers, we have a responsibility to these people to ensure the code is accessible.

If you are in a position to choose your own replacement you may be able to manage this situation. More likely you aren't, the company will go out and hire another contractor. Suppose you give them a few months notice, and suppose they choose a good employee and send him on the appropriate C++ courses. Can he now maintain your program? How many years' experience does someone need to maintain your masterpiece?

As I have observed before, we do not develop in a vacuum, we must be aware on the context we develop in. Are our universal forms truly universal or do they form a barrier to understanding?

Maintenance and reuse

There is not much maintenance in the arts community. Rauschenberg's *White Painting* owned by San Francisco Museum of Modern Art is an exception; the artist has given instructions that this all white piece should, and has been, from time to time repainted.

Software development, like art, is a exercise is almost pure intellectual creativity. But unlike artists we have to produce works that are practical, useful and maintainable.

As an exercise in intellectual effort art has already visited many of the same ideas as software. Dan Flavin, for example, practises component reuse, using standard neon-light tubes (which may be readily purchased by anyone) he constructs pieces such as his Monument to V. Tatlin.

Carl Andre (considered a minimalist) has pieces made in a factory. Consider Maholy-Nagy (a constructivist) "just before he left for the Bauhaus [he] ordered a series of three paintings by telephone, giving a factory that made enamel signs precise verbal instructions and leaving the manufacture up to them[8]."

As software developers we are in this space: we want to use standard components, we want to be able to produce software from precise specifications – some would even argue for a code factory. But, much of the art that arises from these techniques is considered inaccessible, this is not a quality we want in our code. Ironically, minimalist paintings created to be free on context are difficult to understand exactly because of the lack of context!

There is more than one way to skin a cat

Sculpture, minimalism, installation, ready-mades these are the languages of modern art. They allow artists to express their ideas. C++, Java, Python, Perl – these are the languages of software. They allow developers to express their ideas.

We choose a language which is expressive enough to manipulate our ideas, we choose Java for internet applications, Visual Basic for desktop application and so on. None of these languages yield more computing power, they are all Turing equivalent, no more no less. Yet each in its own field is more expressive, the power of expression is what gives one language power over another. The expressiveness means nothing to the machine; it is expressive power to the human developer.

This richness of expression comes at a cost: the size and complexity of the language increases. Wirth's language family has gone down the path of minimalism, yet Oberon and Modula-3 are seldom used outside research environments. Oberon is so minimalist it eliminates the enumeration types and the for-loop, in becoming so minimalist it has removed the expressive power of its users. This pursuit leads to Orwell like *NewSpeak* where it is not possible to think an incorrect thought because the language does not permit it. Instead the successful languages allow expression - and they allow us to make mistakes. Maybe the balance has gone the wrong way; maybe we do have too many features. Ironically many of these features are aimed at allowing reuse. In example 3, code reduction occurs because we use a ready-made algorithm. Yet the price of this is that we must understand the context.

Reuse is the Holy Grail of software development yet we must not forget that the first reuse of our code occurs in the maintenance phase, MegaCalc version 1.1 reuses almost everything from version 1.0. Making code more accessible, that is, more understandable, benefits reuse in version 1.1 and in MegaWord 1.0.

We want to reuse and have added vocabulary to our language to allow expression of reuse. However, growing the language not lead to minimalism, it leads to constructivism because we have increased the amount of context information required to understand the code.

The superfluous has no place in minimalism, constructivism or program code. However, we must consider our audience. We can do this through shared context or through explicit statements. Our shared context, our universal truth, is our language. But this truth is not singular, it is many and we cannot expect even a true believer to know the entire truth.

We can reduce the physical amount of code but if we simultaneously increase the amount of knowledge required to understand it what have we gained?

> "C++ supports the notion of gradual introduction... programmers can remain productive while learning C++[9]."

Of the examples above no one piece of code is superior: they are simply different. They may simply be examples written by developers at different points in their learning cycle. To tell the author of example 1 that she should of written example 3 adds nothing to the code but you may have an adverse effect on her self esteem, her attitude and how she views you.

The place for minimalism

> "The results where shown... in New York in 1966... the artist [Andre, Judd, Morris, Flavin] shared a concern with stripping sculpture down to its essence[10]."

Most developers have rejected minimalism languages. Minimalism at the code level is self-defeating. Constructivist code is both advantageous and dangerous.

The place for minimalism is in our designs. A design should stand alone without distractions. Our design is the kernel of our system as such we should ensure it is extendable. Extending a design should not introduction distractions and complications.

Conclusion

There is nothing wrong with minimalism. I too feel I home with it, I too feel I could have invented it. However we must direct it precisely to avoiding feature creep. We must delve down into the essence of our problem domain and sculpt our solution. We need the expressive power of our language but this brings the responsibility to ensure that we use it correctly.

> "In the development of our understanding of complex phenomena, the most powerful tool available to the human intellect is abstraction[11]."

Constructivism and minimalism are examples of abstraction. They are also tools in their own right which can used to explain and understand software. Our challenge is to keep our products accessible, the blind pursuit of abstraction, minimalism or constructivism is evil when it deprives us of context or burdens us with too much context.

For me, Wassily Kandinsky could have been talking about software development when he said:

> "The 'artist' gives birth to a creation in a mysterious way full of secrets and enigmas. Freed from him, it attains an independent life, becomes a personality, a subject whose spirit breathes on its own but also lives a real material life; who is a being[12]."

How many of us know the life our creations are leading today?

---

[1] Kevlin Henney, Overload 45, October 2001

[2] http://www.artmovements.co.uk/minimalism.htm

[3] http://www.artmovements.co.uk/constructivism.htm

[4] Frank Whitford, Bauhaus, Thames and Hudson, 1984

[5] David Sylvester, *Curriculum Vitae*, About Modern Art, Pimlico, 1997

[6] I originally made this observation on the ACCU-general mailing list in September 2001, Ric Parkin was good enough to provide the original quote " Any sufficiently advanced technology is indistinguishable from magic."

[7] Steve Maguire, Debugging the Development Process, Microsoft Press, 1994

[8] Frank Whitford, Bauhaus, Thames and Hudson, 1984

[9] Bjarne Stoustrup, The C++ programming language, third edition, Addison-Wesley, 1997

[10] Anna Moszynska, Abstract Art, Thames and Hudson, 1990

[11] C. A. R. Hoare, "Notes on Data Structuring", in "Structured Programming", Dahl, Dijkstra and Hoare, Academic Press. Thanks to Rob D'Entremont and Peter S Tillier on ACCU-general for providing the source of this quote.

[12] Quoted at Berkeley Art Museum, 2001