SOFTWARE COMPONENTS

Kent Sandvik/DTS

INTRODUCTION

The idea of reusable components (or software ICs) it's one of the Holy Grails of the computing industry. The idea is to raise the productivity of programmers by using prebuilt software components. Most industry pundits agree that this is a good thing. However in practice there are very few cases where software components are actually used (Next IC packs, MacApp to some degree).

REINFORCEMENT

We should really look at these components as higher level primitives glued together by a programming language. In this way components will constitute a reinforcement mechanism. A new buzzword concerning this notion is the concept of mega-programming, where programs are constructed by gluing together existing functionality blocks.

In general building applications using components is a bottom-up activity. These components could be reused by many other applications. In many cases a shared library system would provide the necessary means to packaging these components -- assuming that this shared library system provides support for object oriented languages, as well as provides support for *any* programming languages (reuse of the component).

Use of components is nothing new under the sun. One reason UNIX and C became very popular has to do with the reuse of the standard C and UNIX libraries. Similarly the Macintosh toolbox could be conceptualized as a big mechano box with toolbox procedures as the building blocks.

ENGINEERING

Looking at other engineering fields the engineers work with pre-existing building blocks (IC circuits, bolts, mechanical parts…). One possible scenario to move software construction towards an engineering disciple is to reinforce the use of existing software building blocks. This would also eventually solve some of the problems with the currently known software crisis. It is a known fact that increasing complexity in software leads to more expensive and difficult solutions. Looking at the world of biology of mathematics we know today that simple and flexible entities that interact with each others are capable of building very complex and robust functionality. In other words complexity appears when the simple and robust building blocks are properly designed.

WHAT IS A SOFTWARE COMPONENT
In general we are looking at a software entity that could be reused as much as possible, rewired in various combinations to provide meta-functionality in combination with other components.

To make a component reusable in many various applications it is highly important that *the component is independent of the application for which it was possibly designed*. There are exceptions, sometimes a component is tied to a particular application of object framework. However it is always a design goal to write components that could be separated from any dependencies.

To use a component means in practice that we can save time because we do not need to know how it works on the inside. We only use it from the outside. A complex component that is easy to use will for the developer. It forms a conceptual simplification of what is implemented.

DESIGN
All this means that we need to carefully design components in order to avoid as many dependencies as possible, as well as design the interface in a programmer friendly way. If possible go for the simplified solution instead for a more complex one. This so the programmer should not spend a lot of time learning how to use the component, instead spend the time wiring the components to achieve more complex solutions.

It also means that the documentation should clearly demonstrate how the component should be used. If possible we should provide many various ways to look at the use of the component (paper documentation, snippets, browser information, annotation tools, and so on).

We should not include too many operations on the component since it makes the component hard to understand and use. If possible the component should capture all operations that make it meaningful to use, but no more.

One possible problem that might appear from this concept is that the component becomes *too generic* -- in other words it does not make sense to write this component.

QUALITY
It is of utmost importance that the developer should trust the component. If even one of the components in a possible package come with a bad reputation, the trust of the user is gone. The components should be even more thoroughly tested than normal code, because they will become building blocks in a bigger abstraction level.

In addition, components should execute fast, and be small and lean in order to avoid any possible design problems due to use of components.

CONSTRUCTION OF COMPONENTS
Here's a list of criteria concerning construction of components:

• Understandability
The component should represent *a well-known* abstraction. Here's both a good and a bad example:
TSoundRecorder - good, provides a good label for a sound recording component
TSndRecordToFile - a more limited label , does it imply that we could only use this component to record to a file?

Also, the component should have a well defined interface, both syntactically and semantically. If two operations in two different components have the same name, they should act in a similar manner. For instance:

TSoundRecorder->Input() - what does this imply, currently has input, does the system has an input device, something else?
TSoundRecorder->HasInputDevice() - more clear, signals a test finding out if the system has a sound input device (note the use of the prefix Has-, similarly we should use prefixes such as Get-, Set-, Init-, Next, Last, Reset, and so on).

• Independence
The component should be independent of surrounding entities. For instance tying the component to a base class object (MacApp TObject or something similar) severely limits the use of the component in other environments. Note that a highly object oriented philosophy leads to this independence.

• General Abstraction
The component should be a general abstraction that is useful in many several applications without undergoing changes (code rewrite, inheritance, subclassing). The components should be standardized concerning name, fault handling, structure and so on (note the use of Apple Miss Manners for C++ use for instance).

MAINTENANCE
Since components will be used for a long time, maintained by many engineers, it is highly important that they are written so that they are easy to maintain. Another consideration is backwards compability. We need to be aware of version control issues. Yet again a good shared library application with version control should provide the abstractions for component building. The most important part of the component is the interface. It must be general and complete to make the component reusable.

For instance a trivial issue concerning the maintenance and reuse are names (class names, member functions, fields). It actually takes a lot of thinking to create good names. One obvious solution to the naming problem is to standardize on certain names (see Miss Manners). Here's list of good member function names for various occasions:

Get (as in GetRefnum)
Set (as in SetPriorityLevel)
Reset (used with iterators for instance to reset the internal values to a known initial state)
Next (next in an iteration list)
Last (last in a list)
IsLast (predicate signaling if something is the last, note the use of prefix Is-)
Has (as in HasSoundInput)
Record, Play, Forward, Backward (note the familiarity with known consumer terms)

THE ROLE OF INHERITANCE
We could use inheritance when creating software components, looking at possible frameworks that generate components at the leafs of an inheritance tree. The inheritance schemes are very powerful when building such libraries. However, the use of inheritance is also very fundamental for the quality of the component system. It also hinders to some degree the notion of independence concerning components, something that should be the ultimate goal with standalone components.

OTHER PARAMETERS
Here's a list of other useful criteria when constructing components:

• Reduce the number of parameters.
Fewer parameters will lead to more cohesive operations that also imply more primitive but forceful operations.

• Avoid using options in parameters:
Any options should be set by the creation procedures, and special operations should be used to change these options. C++ has a noticeable exception in the case of using default parameters for constructors and member functions, as in:

```
class TSoundRecorder :
public:
    TSoundRecorder(short refNum = kRecordToMemory);
…
};
```

This would signal when constructing a TSoundRecorder that if the constructor has no parameter, the constructor will create a class that records sounds to memory, and if a refnum is defined, then the constructor will create an object that records to a specified file. However the use of two constructors would provide this feature as well, so all we save are lines of code.

• No direct access to internal fields

This is the notation of classical object oriented design, just more strongly enforced in the case of software components, where we should have well-defined input/output, and no backdoors to the internal structures (as in the case of ICs).

• Consistent Naming
This is yet again the issue of providing a very coherent component library that is easy to use and should not take long to learn.

SO WHY DON'T WE HAVE COMPONENTS TODAY?
Here's a list of reasons software components have not widely used today:
• *We have very tight projects where the project teams can't afford to write components*
The solution would be to enforce the use of components by having a separate group responsible for producing components for the whole department or company. This is already the case in some companies (IBM). The other solution is to provide accounting metrics that provide positive financial benefits if reusable components are used.

• *It is not invented here*
This is part of the classical 'macho' image amongst super-programmers. Also, the trust to use code written by someone else is an uncertainty factor.

• *We don't have any standards concerning how to write components*
This is an even bigger issue today when we talk about both comparability and cross-platform issues. Actually, we have work in progress concerning this issue (AppleEvents, OLE, OMG).

• *We can't find any reusable components*
Yet again we have the issue of distribution of components. In the case of Apple we have many distribution channels concerning this. Also, AMIX and similar future electronic information exchange systems will provide world wide distribution of software components.

• *Programmers feel productive when they write many lines of code*
Yet again this is more of a psychological issue that makes the project and the programmer feel good based on N lines of code (where N is a very huge number, the bigger the better). To some degree this is actually a very bizarre situation, the programmer should spend time solving hard problems, and not the simple ones. This is really a waste of programming resources.

• *Nobody wants to pay the cost of writing software components*
This means that a particular group would write software, which is then reused by another department for free. Yet again this is very narrow minded thinking. For instance in the case of DTS we are in the business of providing sample code, and a slight change to write utility code and components that gain both our developers and the company would be a very rewarding long term prospect.

OTHER HINDERS

Another reason components have not been used have been the issue of using procedural methologies and languages to construct code. Usually when a particular requirement needs a solution, the obvious design methology is to write a function that resolves this issue. Encapsulation and reuse has not been widely used due to the limited design style of procedural programming and data flow analysis. There are exceptions where functions are wrapped around calling conventions (AppleEvents, OMG). However in general such code is in many cases very specialized, and could hardly be reused in other designs.

ANCIENT EXAMPLES

As noted earlier we do have examples of reusable components (UNIX, C). Another good example is the amount of FORTRAN based libraries that float around on the networks. Another programming world where components are the norm is the Smalltalk programming world. AMIX already sells Smalltalk components for a fee between $US5 and $US400.

ADDITIONAL READING

Object Oriented Software Engineering   Jacobson
Object Oriented Sofware Construction   Meyer
Object Oriented Programming   Cox