COSC345 Week 20 Notes

Be careful what you wish for, you may get it.

For many years, "reuse" was the Great Hope of the software engineering world. If only we could build our programs out of ready-built off-the-shelf components, it would be so much easier to write programs. Brad Cox, the inventor of the Objective C language that is used to good effect in Mac OS X and iOS, coined the phrase "Software ICs". At times it seemed as if everyone was talking about components. There were in fact two hopes. One was that we could make programs (and money) faster by using other people's components than by writing our own, and the other was that we could make money by writing components for other people to use. Admittedly, this was always going to be hard for languages like Fortran 77 (but not Fortran 95, still less Fortran 04), Pascal, C, and so on. But another great hope was the modular languages (especially Ada) and object-oriented languages (like Eiffel) would save the day.

To a large extent this has come true.

If you are a Perl programmer, there is a vast library of things you can reuse called CPAN, the Comprehensive Perl Archive Network, at www.cpan.org.

If you are a Tcl/Tk programmer, there is a web site (www.tcl.tk) where you can start looking. Again, there is a great deal.

If you are a Python programmer, www.python.org is there for you, and www.pythonware.org, and others.

If you are a Squeak Smalltalk programmer, there's such a range of stuff you can get that there is a special browser in the environment to help you find it.

If you are a Java programmer, you know how you are awash in JavaDoc, and the Java Community Process is constantly coming up with new JSRs, some of which turn into standard parts of Java (like Doug Lea's multithreading extras) and some of which are intended to be specifications for businesses to make money by implementing. And of course you will know about all the cool stuff at www.apache.org.

If you are a statistician, or anyone doing complex things with numerical data and wanting to display the results, you really cannot go past R (www.r-project.org). And there is a Comprehensive R Archive Network (click CRAN in the navigation bar), with packages available for more kinds of analysis than most expert statisticians have ever heard of. (Some of them are econometrics, some of them are AI, some of them are bioinformatics, *etc.* In fact there's even the BioConductoR project at www.bioconductor.org which is dedicated to bioinformatics with R.)

If you are writing in {AMS,}{La,}TeX, the Comprehensive TeX Archive Network at www.ctan.org is for you.

If you are a Haskell programmer, the Hackage collection of library modules at hackage.haskell.org currently has over 1000 packages for download (and most packages have more than one module).

All of those examples are community systems built around an open source project (Java *used* to be an exception, but is no longer). But then there's IBM's Alphaworks site.

These days, your average operating system comes with a huge range of stuff you can use. The official UNIX core has over 2000 named items you can use, not including X11, and not including things like OpenSSL or OpenGL or libmkimod (portable sound library) or libpng or ... Windows 5 had over 8000, and that's

not including anything like the Microsoft Foundation Classes. As for Mac OS X, there are frameworks piled upon frameworks. I haven't even managed to *find* all the free Mac OS X documentation yet, let alone *read* it. There's even a toolkit (Dashcode) for writing mini-applications (Dashboard widgets) in HTML + CSS + Javascript that ties in with WebKit. Linux has GNOME (with its Bonobo component model) and KDE (with its KParts component model). Windows has COM. Even entire applications can be treated as components: Word can be extended by programming in Visual Basic for Applications on Windows (but not on Mac OS X any more, although it used to work well) or by programming in Applescript on Mac OS X (but not on Windows). But it can also be driven from outside through a component interface, using a scripting language like Applescript or Fscript on a Mac or Powershell on Windows.

To give you an example of where we've come with reusable software, if I wanted a program that would take a picture every 10 minutes and send it to another machine, these days it could probably be done in about a page of Applescript driving Photo Booth to take the pictures and Mail to send them. Think of the things you could do by combining existing software using tiny amounts of scripting. (The people developing the Android software for mobile phones have...)

We have entire frameworks developed to support component programming and reuse. Visual Basic is nothing else but. The Wikipedia article http://en.wikipedia.org/wiki/Third-party_software_component gives Visual Basic the credit for creating the first real commercial market in third-party components. (Most of the examples above relate to Open Source components or typically come with the operating system.) JavaBeans an attempt to do the same thing for Java. OSGi is related. Of course one of the aims of the .NET Common Language Runtime is to enable components written in any .NET language to be used from any other .NET language, and it seems to be working.

A good overview of components can be found in the Wikipedia article http://en.wikipedia.org/wiki/Software_component#Software_component

In short, we have reuse beyond our wildest dreams.

It used to be that the problem was that we didn't have reuse. Now the problem is that we do. This lecture tries to take a balanced look at reuse.

The first problem with reuse is that you have to *find* something to reuse. That means that you have to push your design far enough to get a clear idea of what you are looking for. In most cases, what you are looking for won't be *exactly* what you want. Either you will have to adapt the thing you've found (perhaps by editing it, or perhaps by writing extra "glue" code around it), or you will have to adapt your design.

This actually comes with three subproblems:

1. *finding* things in an overwhelming sea of documentation. This is an Information Retrieval problem. You really want to put the available documentation into an information retrieval system like Andrew Trotman's atire, or Melbourne University's zettair, or Apache Lucene.
2. *verifying* that what you have found is what you are looking for. This means you actually have to read the stuff. Well-structured documentation follows the "inverted pyramid" style where the most important information is at the front, less important information (such as limitations or dependencies) follows, and actual interface details come last. http://www.ch-werner.de/javasqlite/ is a nice example of JavaDoc written in this style. This style lets someone frantically

wading through a lake of documentation quickly eliminate false leads.
3. *evaluating* which of several alternatives looks most promising. At this point you need interface details so that you can assess the likely cost of adaptation.

The second problem is that just because a "reusable" component exists, that doesn't mean it ever worked, still works in some environment, still works in your environment, or can meet your performance requirements. You still have to do your own tests. Reuse saves development time, but not that much testing time.

The slide "mechanisms for reuse" describes different ways you can reuse something. Basically, it recapitulates the history of programming.

Reuse by linking refers to the 1950s technology of gluing a Fortran program together out of separately compiled subprograms. It also refers to the ability to load new classes into Java or C#. This is just using something without any kind of adaptation.

Reuse via text editor refers to the "make a copy of something close to what you want and change it" approach. You might think that this is hopelessly old-fashioned, but whenever somebody finds an example on the Web, drops into into Eclipse or Visual Studio, and starts hacking, that's reuse via text editor. It makes a lot of sense, except that you lose the link between the original version and the revised version. UNIX programs like `diff -c` and `patch` try to help here: if you have changed version A into version B, and someone else has fixed bugs in version A to produce version C, then maybe (version B + (changes from A to C)) will give you something close to a less buggy version B. Of course, if the changes overlap, it may be hard to figure out what to do. The darcs distributed version control system (see darcs.net/) is based on a "theory of patches", which may help here.

Reuse via macro processor is an attempt to make things that don't need to be edited, because likely sources of variation are already dealt with by parameters in a template which you instantiate using a macro processor. My personal favourite macro processor is m4. I provide an example of reuse via macro processor. Since Java 1.5, Java has had classes with type parameters, so we can now talk about `ArrayList<String>` and `ArrayList<Date>` instead of just `ArrayList`. However, it's an essential property of Java that we cannot possibly talk about `ArrayList<int>`, because `int` is not an object reference type. What you have to do is to use ArrayList<Integer> and rely on Java's automatic boxing (converting an `int` to an `Integer` by allocating a box and stuffing the `int` into it) and unboxing (converting an `Integer` to an `int` by calling `.intValue()` on it) to provide the illusion of storing `int`s. Since Java type parameters do not, in fact, eliminate the type casting operations we used to need in Java 1.4, only hide them, the costs of using box types (like `Integer`) are heavy. So I demonstrate a template "al.m4" which you can use to build things that look exactly like `ArrayList` but are specialised to any type you want. The performance boost is dramatic. And it cannot be had any other reasonable way.

Reuse via generics (Ada, Eiffel, and Java term)/templates (C++ term) is basically about getting the compiler to do the template instantiation. (The same idea is also present in ML, CAML, F#, Clean, Haskell, and Mercury, in a somewhat different form.) The designer of a reusable software component has to predict which aspects might vary and express those aspects as parameters, which the compiler can then fill in. The great advantage here is that the compiler understands the semantics of the language, which m4 does not. There is a limitation: some things you can do with a macro processor cannot be done this

way.

Reuse via generator leads us into the area of Domain Specific Languages, and the idea of getting a *program generator* to generate code in a low level language like Java or C# from a higher level specification. There's a paper on applying aspect-oriented programming to sparse matrix algorithms that explains this clearly. Some truly amazing things can be done this way. What is reused here is not source code, but techniques for creating source code. When I've had to write code with a repetitive structure, but not identical repetitions, I've often used AWK to do it. Other people would use Perl or Python, it's just that I know AWK much better.

One important issue of course is that it's not just source code that can be reused. Architectures can be reused. There are plenty of good books about software architecture. One that is easy to follow is the "gang of five" book about design patterns. (Not the gang of four, the gang of "five".) Designs can be reused. Documentation can be reused. Forms can be reused. Even if tests cannot be reused, perhaps test case generators can.

Reuse remains a worthy goal, but developing for reuse is harder than normal development. The documentation for reusable components had better be really good. Examples of reuse that can be imitated are especially important. It is important to make reusable items *findable* otherwise they will not be reused. (For example, one of the reasons that the programming language Clean never became as popular as Haskell, which it greatly resembles, is thought to be the fact that it is so hard to find using an internet search engine. I don't really buy that, since "Clean programming language" as a Google query puts the right web site at the very top of the list of results. But it makes a nice cautionary tale.)

Even though you got a component from someone else, it is still your responsibility to test the aspects of it that you use. Just because someone *says* a component is correct or portable doesn't mean it *is*, even if they charge you money for it. As several groups found this year, a well-recommended component may still have defects, and while it wasn't their *fault*, it was still their *problem*, either to fix the defects or to demonstrate that their code did not exercise the defective parts. The company I worked for found out the hard way that it can be more expensive to use a third-party component than to develop your own.

Think about it differently. Instead of thinking about reusing someone else's *code*, think about using their *knowledge*. For example, converting a string from upper case to lower case is shockingly hard in Unicode. (The result is not necessarily the same length as the input, and what the result should be depends on the locale.) A good reason to use the International Components for Unicode from IBM is not because of their (presumed) skill at *coding* but because of their (presumed) understanding of the depths of Unicode weirdness.

Of course, if you don't know enough to *write* a component, you may not know enough to *test* it either. That's OK, you may be able to find some test cases to use!