Struggling with a C++-based component architecture

by

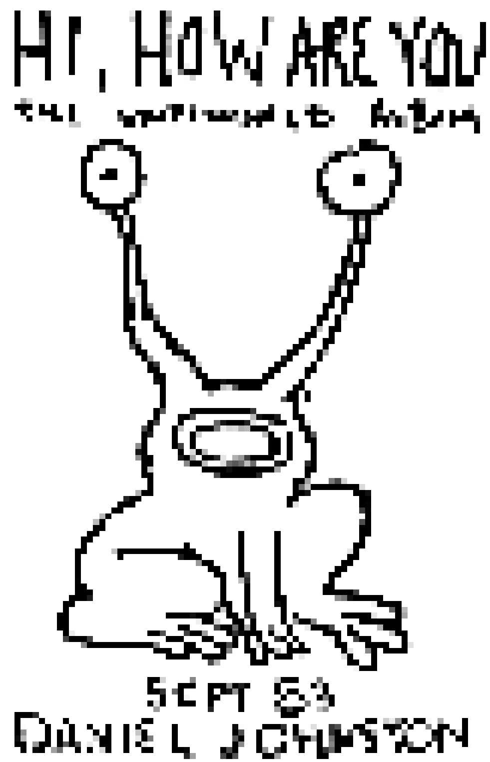
Stephan Bergmann
sb@openoffice.org

Agenda

- About the speaker
- The UNO framework
- What is a component
- C++ vs. C for components
- The trouble with inline
- Hidden inlines
- Case study: weak symbols
- Case study: exception classes
- Case study: vtable relocation
- Dependencies
- Conclusion/questions

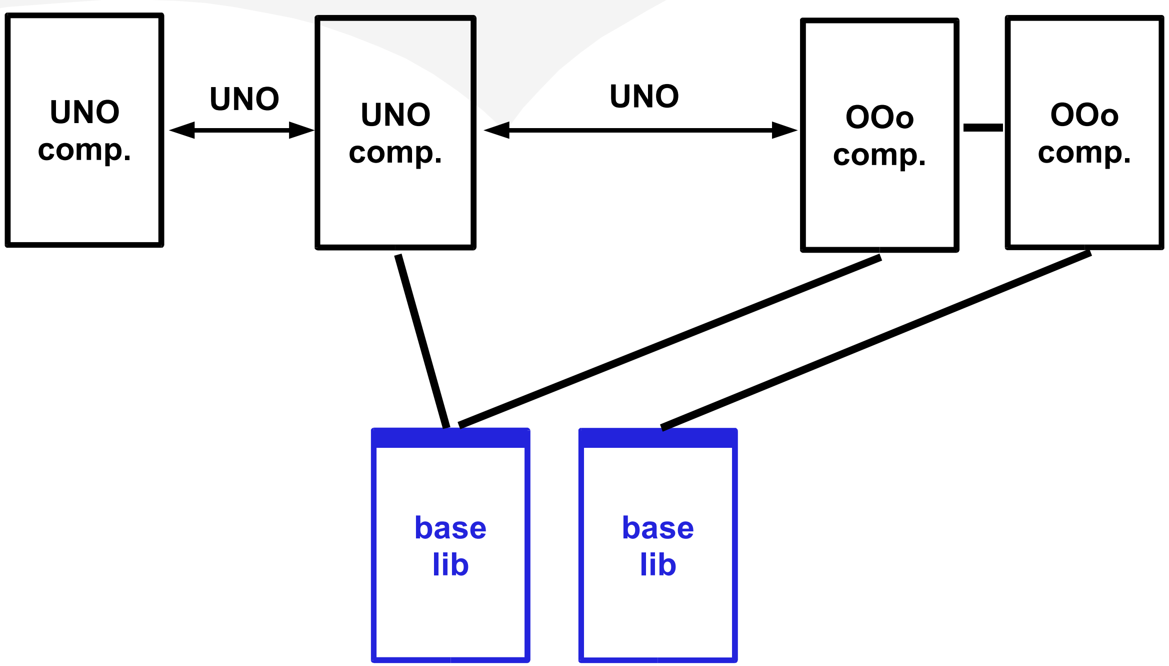
About the speaker

- Stephan Bergmann
- Software Engineer, Sun/StarOffice
- Base technology development (formerly UCB, now UNO; text encodings, URLs; C++, Java, standards)



The UNO framework

- UNO is a component framework
- Broadly, three sorts of components are involved:
  - UNO components: stable interface defined in UNO IDL; implemented in any language
  - Base/helper libraries (e.g., sal, cppuhelper): stable interface, written in C/C++
  - OpenOffice.org components (e.g., sw, sfx2): fast-changing interfaces, written in C++



What is a component

- Abstract view: cohesive set of functionality
- More practical view: a set of C/C++ header files
- Practical view: the implementation behind the set of C/C++ header files
  - Either a shared library
  - Or nothing, in case of a set of purely inline C++ header files

C++ vs. C for components

- C++ has advantages over C, for both the interface and the implementation of a component:
  - “Resource Acquisition is Initialization”

```
{
   osl::MutexGuard aGuard(aMutex);
   ....
}
```

vs.

```
osl_acquireMutex(aMutex);
...
osl_releaseMutex(aMutex);
```

  - Exceptions (containing information) instead of error returns and global errno

```
try {
   something();
} catch (BadProblem & e) {
   display(e.what());
}
```

vs.

```
if (!something())
   display(strerror(errno))
```

  - Language takes burden of many clerical tasks from programmer (implicit constructors, destructors, ...)
    In interfaces, this higher level of abstraction and ignorance of low-level details can lead to problems

The trouble with inline

- Inline across component boundaries increases coupling:
  - One component suddenly depends on implementation details of another component (exposed through inline code)
  - Fixing the implementation of a component requires re-compiling other components for fix to take effect
    Global consistency in danger (e.g., calculating hash values)
- Code and symbol table bloat
- Never use inline in a component interface (and beware of hidden inlines)
- Even purely inline C++ header files can cause trouble (e.g., rtl::OUString)

Hidden inlines

- Hidden inlines include:
  - Compiler generated class members: default and copy ctors, dtors, assignment operators (for ctors and dtors, both complete object and base object variants)
  - RTTI (exception handling): typeinfo structs, typeinfo name literals
  - vtables: vtables themselves, deleting dtors, thunks, RTTI
  - Compiler optimizations: allocating ctors, deleting dtors

```
struct A { virtual ~A() {} };
struct B {
   virtual ~B() {}
   virtual B * clone() { return new B; }
};
struct C: public A, public B {
   virtual C * clone() { return new C; }
}
B * f() { return (new C)->clone(); }
```

g++ 3.0.1:
26 weak symbols

Case study: weak symbols

- Vague Linkage:
  - C++ compiler sometimes does not know where to emit code/data (e.g., inline functions, vtables). Solution: emit code/data everywhere it is needed, as weak symbols.
- Semantics of weak symbols at runtime:
  - When the dynamic linker searches for a weak symbol, and does not find a definition, it uses null instead
  - On GNU/Linux, the dynamic linker favors strong over weak definitions At runtime, this slows down relocation of weak, defined symbols used for vague linkage
  - Test: change weak, defined symbols into global ones:

```
> time ./soffice.bin
                    1.310u > 1.000u (~75%)

> (LD_DEBUG=symbols,bindings ./soffice.bin 2>&1) | wc -l
                    1529943 > 1152978 (~75%)
```

Case study: exception classes

- How to write an exception class?

symbols (def+undef) needed when catching/throwing:	wntmsci9	unxsols4	unxlngi4	g++ 3.2
Class OpenEx {};	**1**+0/**4**+0	**1**+0/**1**+0	**2**+0/**2**+0	**3**+0/**3**+0
Class MediumEx {				
public:				
MediumEx();				
MediumEx(MediumEx const &);	**1**+0/**4**+3	**1**+0/**1**+2	**2**+0/**2**+3	**3**+0/**2**+3
~MediumEx();				
MediumEx & operator =(MediumEx const &);				
};				
Class ClosedEx {				
public:				
ClosedEx();				
ClosedEx(ClosedEx const &);	**1**+0/**4**+3	**0**+1/**0**+3	**2**+0/**2**+3	**1**+1/**0**+4
virtual **~ClosedEx();**				
ClosedEx & operator =(ClosedEx const &);				
};				

Case study: vtable relocation

- Shared library code calls function f directly (lazy relocation, cheap at startup):

```
                call f@PLT

.PLT0:          pushl 4(%ebx)                     .GOT:      long reserved
                jmp *8(%ebx)                                 long lib identifier
                ...                                          long dynamic linker
f@PLT:          jmp *f@GOT(%ebx)                             ...
                pushl offset                      f@GOT:     long f@PLT + 1
                jmp .PLT0
```

- Shared library code calls function f through vtable (eager relocation, expensive at startup):

```
                movl (%eax), %eax
                addl f@vtable, %eax
                movl (%eax), %eax
                call *%eax

vtable:         long reserved
                ...
f@vtable:       long f            < resolved at startup
```

Dependencies

- C code:
  - Only depends on (stable) C library (“libc”)
- C++ code:
  - Also depends on (stable) C library (“libc”)
  - Depends on (emerging) C++ library (containing functions for exception handling, RTTI handling, memory management, object/array con-/destruction, pure virtual function calls, ...)
  - Depends on STL:
    STL use often produces inline code (i.e., no runtime dependency), but some code is non-inline
    Use of STL types in interfaces introduces dependency on STL's internals
  - Dependencies on C++ library and STL have implications for mixing components

Conclusion

- Using C++ instead of C in a component architecture has advantages at the language level, but it also introduces practical problems:
  - Naïve use of C++ degrades performance (e.g., startup performance through symbol table bloat)
  - Some C++ language constructs have bad implications for components (e.g., inline increases coupling)
  - Using C++ for shared libraries has not matured yet (e.g., eager resolution of vtable function symbols)
  - C++ code has broader dependencies on runtime environment than C code
- The struggle goes on...

Links & questions

- OpenOffice.org http://www.openoffice.org
- Subprojects http://udk.openoffice.org, http://porting.openoffice.org/

Questions?