# Bridging C++ to Python and vice-versa

Jack Jansen
Centrum voor Wiskunde en Informatica

# About Me

Active Python user since 1992,
"responsible" for MacPython since 1995,
when Guido got rid of his Mac.

Work at CWI on multimedia research.

Also: father, bass player, long time Unix devotee, punkrocker, hiker, party animal, anarchist, etc

# About the Talk

- Problem definition
- Python and C++ Object models
- Bgen
- Extending bgen for C++
- Closing remarks

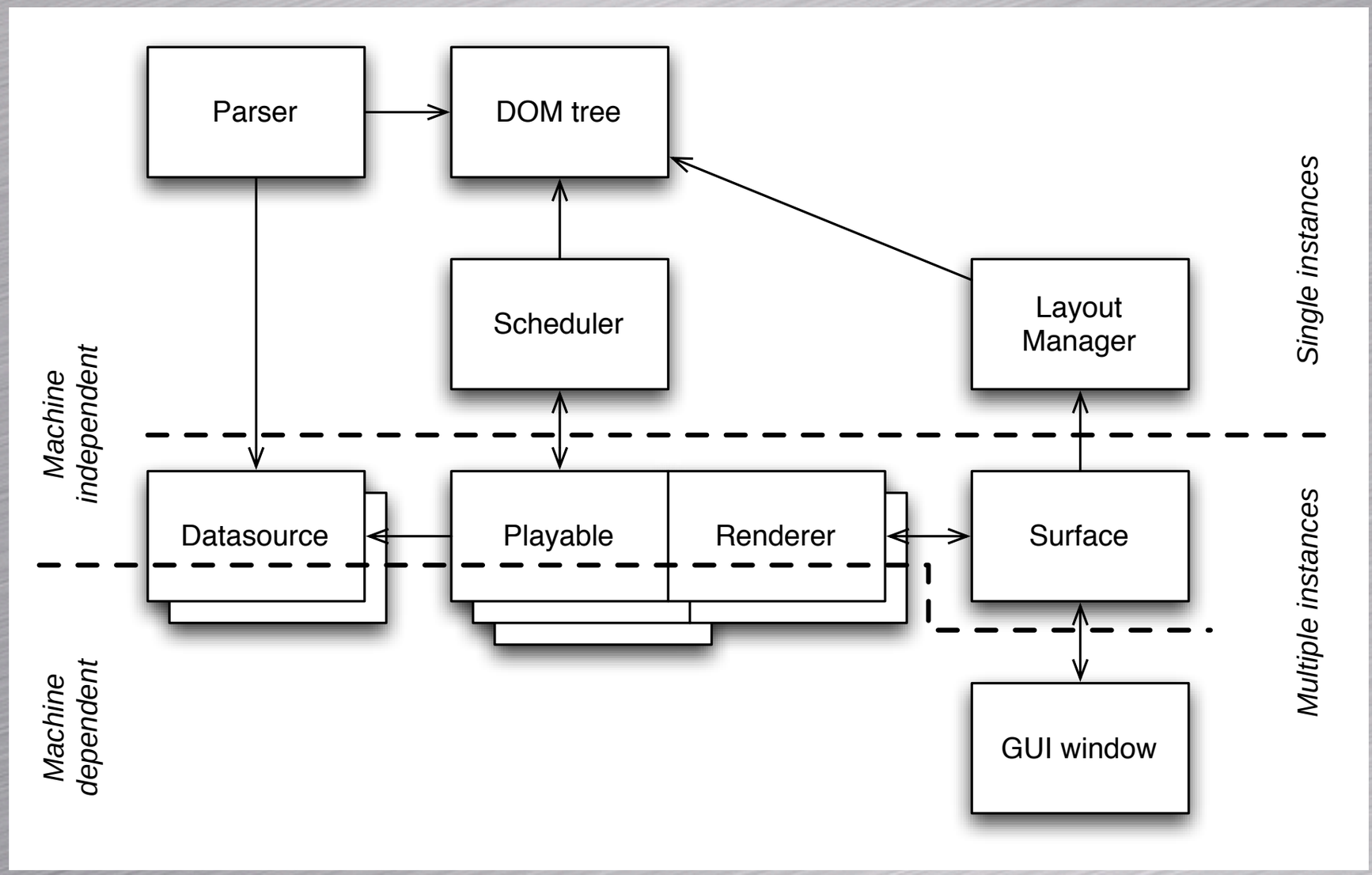
# Defining the Problem

- We have: extensible multimedia playback framework in C++
- We want: extensible multimedia playback framework in Python
  - It makes experimenting soooooo much easier...
- Therefore: 2-way cross-language calling and subclassing

# Ambulant Goals

- Extensible multimedia playback engine in C++ (but you knew that already:-)
- Targeted at research/development
- Portable (Linux, Win32, WinCE, Mac)
- High-performance
- **ALL** components replaceable and extensible

# Ambulant Design



# Sample API

```
/// Determine where to render media items.
/// The layout_manager manages the mapping of media items to regions.
class layout_manager {
  public:
        virtual ~layout_manager() {};

        /// Return the surface on which the given node should be rendered.
        virtual surface *get_surface(const lib::node *node) = 0;

        /// Returns the image alignment parameters for the given node.
        virtual alignment *get_alignment(const lib::node *node) = 0;

        /// Return the object that will receive notifications when the given node is animated.
        virtual animation_notification *get_animation_notification(const lib::node *node) = 0;

        /// Return the object that the animator will control for the given node.
        virtual animation_destination *get_animation_destination(const lib::node *node) = 0;
};
```

# C++ Object Model

- Pure virtual classes are really interfaces
- Overloading of methods (and constructors) by argument signature
- Not needed, yet:
  - Operators
  - exceptions
  - templates

# Python Object Model

- PEP252 and PEP253 are the definitive reference
- 3-step creation with:
  - tp_alloc(type) to allocate storage
  - tp_new(type, args) to create blank object
  - tp_init(obj, args) to initialize object
- Allows 2-way subclassing between Python and C

- `obj = MyType()`
  - To create an object: call its type (`tp_call` on its metatype)
  - `PyType_Type.tp_call` first calls `MyType.tp_new(MyType, args)` then `obj.tp_init(args)`
    - `MyType.tp_new` calls `MyType.tp_alloc`
    - All calls through the normal inheritance scheme

# The Problem Revisited

- Python object implementation that has reference to C++ object, `tp_new` calls constructor, method calls forwarded
- C++ class that implements interface, has `PyObject` pointer, constructor does `tp_new`/`tp_init` dance, method calls forwarded
- Glue between the two

# Bridging C++ to Python

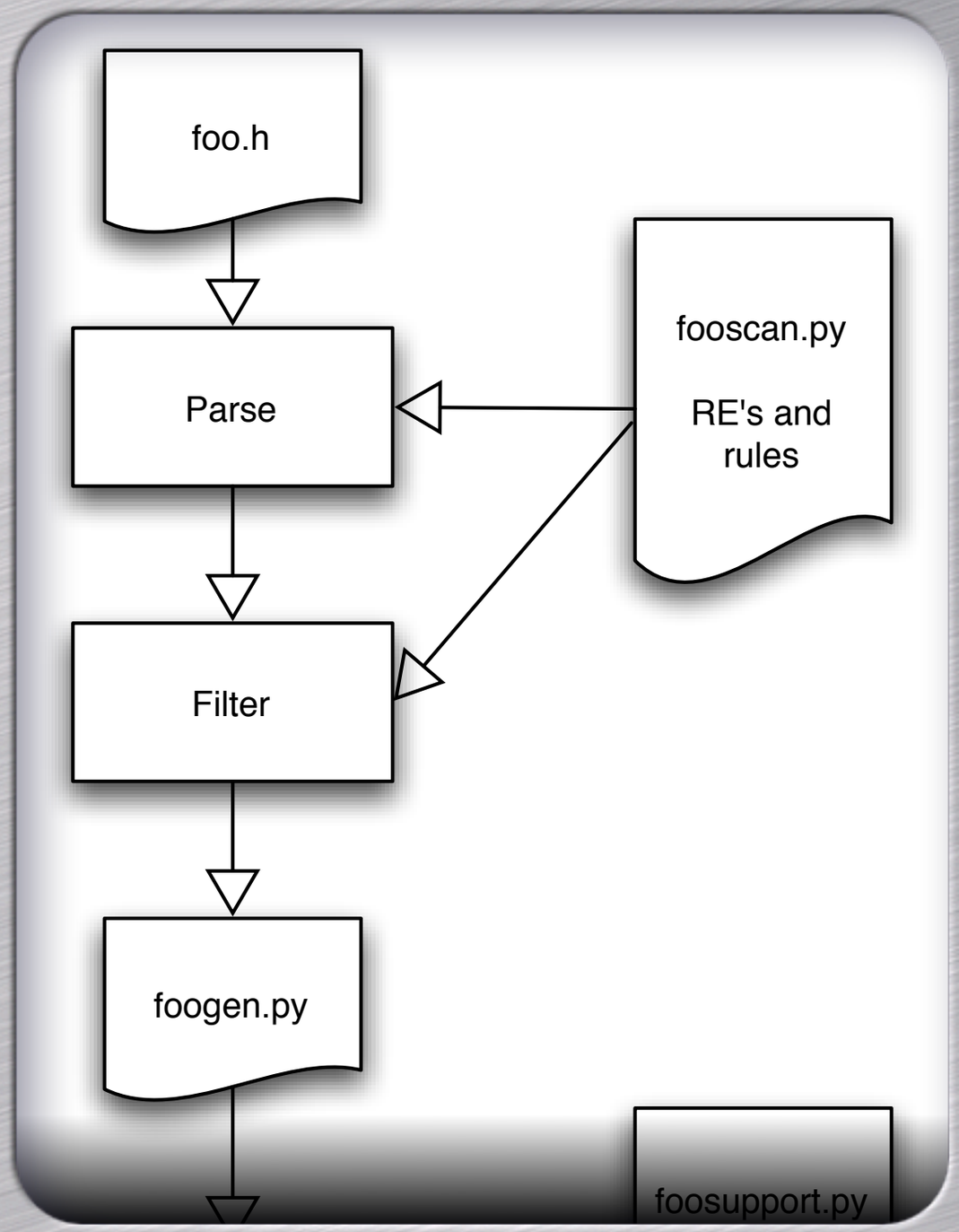
- Hand-coded is not an option
  - even with Pyrex
- Swig is gruesome
  - Lots of Python glue, generated code unreadable
    - difficult debugging
- Boost is huge
- SIP?

# Bgen history

- Written by Guido, maintained by me for 10+ years
- Builds Python interfaces to Mac toolbox modules (Quicktime, Carbon, CoreFoundation)
  - parses 100K lines of C header files
  - generates 30 modules containing 3000 methods (85K lines of C) and 10K constants

# Bgen Control Flow

- Parse .h file with set of RE's
- Transform arguments to be more pythonesque
  - `const char *buf, int size` becomes Python string
- Filter methods and functions by argument signature



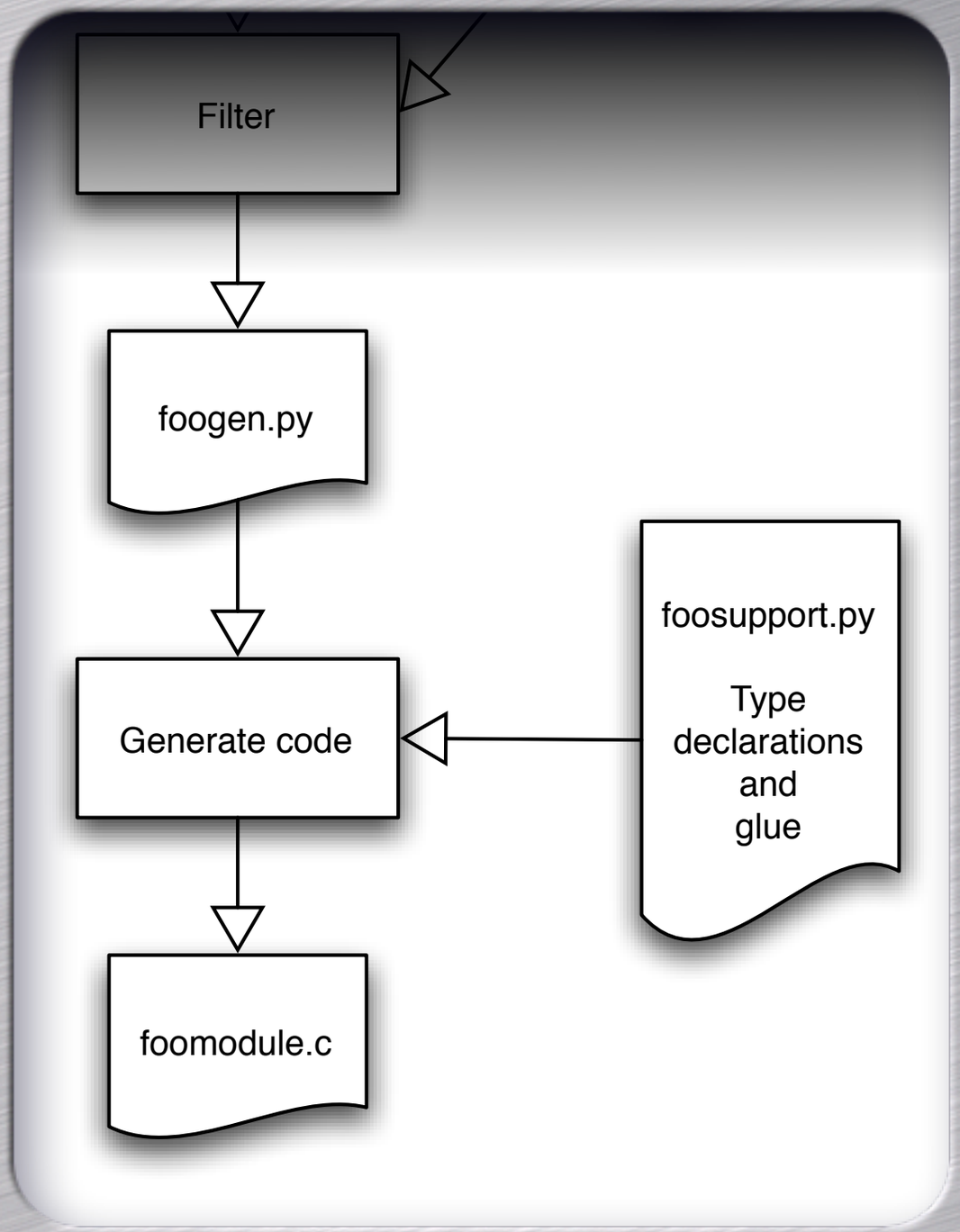
# Bgen Sample

```
EXTERN_API( void )
HideDialogItem(
   DialogRef          theDialog,
   DialogItemIndex    itemNo)                    ONEWORDINLINE(0xA827);

EXTERN_API( Boolean )
IsDialogEvent(const EventRecord * theEvent)       ONEWORDINLINE(0xA97F);
```

```
f = Method(void, 'HideDialogItem',
    (DialogRef, 'theDialog', InMode),
    (DialogItemIndex, 'itemNo', InMode),
)
methods.append(f)

f = Function(Boolean, 'IsDialogEvent',
    (EventRecord_ptr, 'theEvent', InMode),
)
functions.append(f)
```

# Bgen Control Flow - 2

- Intermediate output is Python code
- Supply "type declarations" to handle argument conversion
- Generate code



# Bgen Sample Output

```
static PyObject *DlgObj_HideDialogItem(DialogObject *_self, PyObject *_args)
{
    PyObject *_res = NULL;
    DialogItemIndex itemNo;

    if (!PyArg_ParseTuple(_args, "h",
                          &itemNo))
        return NULL;
    HideDialogItem(_self->ob_itself,
                   itemNo);
    Py_INCREF(Py_None);
    _res = Py_None;
    return _res;
}

static PyObject *Dlg_IsDialogEvent(PyObject *_self, PyObject *_args)
{
    PyObject *_res = NULL;
    Boolean _rv;
    EventRecord theEvent;

    if (!PyArg_ParseTuple(_args, "O&",
                          PyMac_GetEventRecord, &theEvent))
        return NULL;
    _rv = IsDialogEvent(&theEvent);
    _res = Py_BuildValue("b",
                         _rv);
    return _res;
}
```

# Bgen Parser for C++

- Better parser: classes, namespaces, inline functions, default arguments, ...
  - But still doable with RE's!
- No need to filter methods/functions, C++ has real classes

# Overloading

- Need to handle name overloading
  - Only for constructors, today

```
if (PyArg_Parse(_args, “i”, &ival)) {
    // body for method(int ival)
} else
if (PyArg_Parse(_args, “s”, &sval) {
    // body for method(const char *sval)
}
```

# C++ to Python

- Simply rerun intermediate Python code with completely different generators!
- Argument handling and return value handling more-or-less reversed

# Future Work

- Bgen really needs a makeover
  - Packages exist since Python 1.5 or so:-)
  - Visitor paradigm seems very useful
  - Distutils-based?
  - “Interactive” RE generation?
  - Volunteers?
- Bridging exceptions, operators

# Questions?