Generative and Incremental Approach to Scripting Support Implementation

Vespe Savikko
VTT Information Technology[*]
P.O.Box 1206, FIN-33101 Tampere, Finland

Abstract

Many systems may benefit from scripting support, but the implementation of it is seldom trivial, especially if the system has not originally been developed with scripting support in mind. In this paper we describe a generative, incremental process for creating an intuitive Python interface to a large, hierarchic COM library. The approach is illuminated with the original, real-life case study.

Keywords: generative programming, incremental development, scripting, Python, hierarchic COM library

1 Introduction

Scripting languages are often used to add scripting support to existing software written in non-scripting (compiled) languages. The level of support can range from embedding an interpreter into an application to interfacing a library with a scripting language. The benefits of the scripting support are numerous: powerful language constructs combined with higher level abstractions make the use of underlying software easier and interactive interpreter sessions give the programmer a finer level of control for exploration, debugging and prototyping. However, the realization of these benefits depends on three factors: the underlying software, the design of the scripting support and the chosen scripting language. The same factors naturally affect the implementation of the scripting support as well.

In this paper we present a process and an architecture for implementing a Python [1] interface to a large COM (Component Object Model) library, thus providing easier mechanisms for manipulating the objects of the library. We use Python both as a tool for generating constituent parts of the architecture and as a tool for accessing the COM objects. Python was chosen for the scripting language because of the following reasons:

Dynamic object model. Python is an object-oriented programming language but it does not emphasize data hiding. Instead it is possible to write class code that analyzes and manipulates its own state even if the instance is actually a subclass instance (see Section 4).

`eval`. Interpreter services are available within Python code as well (see Section 4).

PythonCOM. Python has an excellent COM support in the form of PythonCOM [2]. However, PythonCOM cannot directly interface with COM libraries that lack Automation and friendly parameter types (see Section 3).

Tool building. As a scripting language Python is very well suited for tool building. Our process uses Python scripts extensively.

The process is best applicable to libraries that define a hierarchic taxonomy of classes with a limited set of functionality (most of interfaces contain only mutators and selectors). The resulting Python interface has the following benefits:

- The implicit taxonomy is mapped to intuitive class hierarchy.
- The mutators and selectors are replaced by a properties mechanism, where instead of methods a uniform protocol for setting and getting values is used.
- The interface for taxonomy parts is mostly generated.
- The process supports incremental development where only the needed parts of the COM library are wrapped. In terms of adaptability this means that the class hierarchy adapts to new information by including the newly generated classes in each process cycle.
- The interface integrates seamlessly with other Python code thus hiding its COM origin.
- Naturally the process independent benefits of the scripting support still apply (e.g., interpreter sessions).

[*]Major part of this work was done in the Software Systems Laboratory of Tampere University of Technology

We describe the process with the original case study which is a non-trivial scripting support implementation: TEDious Scripting is a Python interface to a software design environment TED [3]. The case study approach allows us to discuss the concrete issues of the process and give a solid, experience-based view of its applicability. The stage is set with a process description in Section 2. The problem domain (the relevant features of TED) is described in Section 3. The TEDious Scripting architecture and some of its implementation issues are discussed in Section 4. Finally we wrap things up with evaluation of TEDious Scripting in Section 5 and conclusions in Section 6.

2 Process Overview

The general idea behind the process is quite simple: a large hierarchic COM library is probably easier to manipulate with an intuitive Python interface (with properties and class hierarchy) than with C++ using COM. Since the creation of the interface cannot be fully automated it is useful to have a process that automates some of the more mundane tasks and supports incremental development so that the wrapping can proceed in the order of necessity.

Figure 1 shows the main components of the process and its products. The runtime architecture of the system (which is the product of the process) consists of three layers: original COM library, adapter COM library that wraps the contents of the original library and the Python interface that is the only part directly visible to the user.

For the taxonomy parts of the original library, the adapter library counterparts are incrementally developed and partly generated with the help of ATL (Active Template Library) Object Wizard and Python scripts. The taxonomy classes of the Python interface are freshly generated at the end of each process cycle. A process cycle can be summarized with the following steps:

1. The interfaces to be wrapped are selected from the original IDL (Interface Definition Language) files and the corresponding adapters are created with ATL Object Wizard.

2. A Python script is executed that updates the contents of the adapter library source and IDL files to reflect the additions.

3. Those methods of the new adapter classes that were not automatically created in the previous step are written manually by the developer.

4. According to the adapter library IDL files Python classes (with properties and hierarchy) are generated by a Python script.

Our experiences with the development scripts have been so good that we can wholeheartedly agree with Tip 29: [4]

> WRITE CODE THAT WRITES CODE
> Code generators increase your productivity and help avoid duplication.

Even though Python was the natural choice for us, the ease of building the scripts was still somewhat surprising. Since the structure of IDL files is very systematic, Python's support for regular expressions was sufficient and no specific parser was needed. The scripts also have lots of common code. Much of this code sharing was not originally designed but provided by Python's flexible module concept, where functions of a module are seen by other modules.

One aspect of the development is the documentation. In TEDious Scripting, HTML (HyperText Markup Language) documentation is generated from Python source code. As a generator we use a modified version of Ka-Ping Yee's `htmldoc` toolset[1]. Our modifications include a simple support for packages and Structured Text (of the Zope[2] fame). Also the properties of the wrapper classes are shown in the documentation.

3 TED

TED (Tde EDitor) is a multi-user systems design application created at Nokia Research Center. The system is a full-fledged design environment with graphical user interface, UML (Unified Modeling Language) support, collaboration features and shared repositories (databases). In addition, TED also provides an API (Application Programming Interface) that allows a developer to manipulate a repository directly making it possible to perform quite complicated tasks, like model analysis, various manipulations and transformations, and perhaps even extending the environment with features like model, report and code generators. Since TED's target platform is Windows, the API is implemented as a COM server.

While the TED API is quite extensive (much of UML metamodel is represented with it) its structure, related idioms and the lack of comments have inhibited its convenient usage even with the favored implementation language C++. In the context of this paper the TED API can be seen as an example of a legacy COM server that is written in a C++-centric fashion and its UML metamodel parts create a taxonomy as described in Section 1.

The C++-centricism is evident in the IDL files that define the classes and interfaces supported by the TED API. Generally a designer of a COM service has a couple of mechanisms at his/her disposal that can make the

[1] `http://www.lfw.org/python/`
[2] `http://www.zope.org`

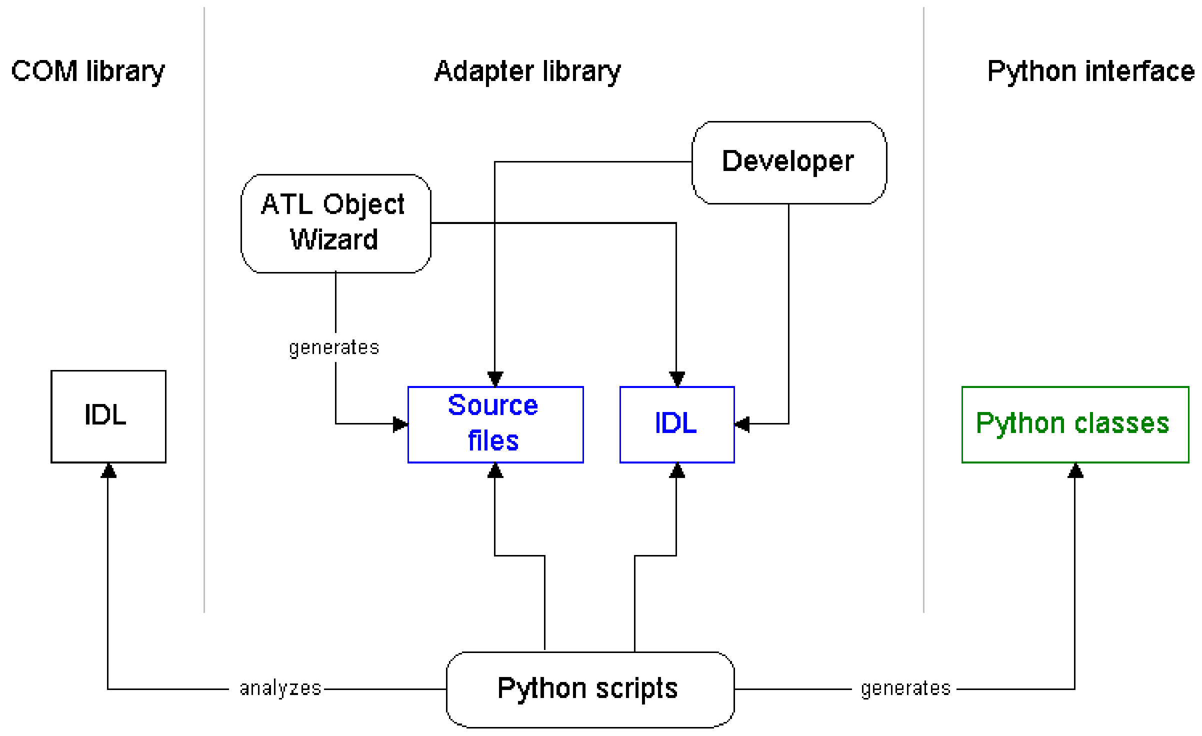

Figure 1. The main components and the products of the process. The unmarked lines refer either to updating or analyzing existing files depending on the state of the process. For example, the Python scripts both update the adapter IDL files and use the same information to generate the Python classes.

service usable from the plethora of programming languages that support COM. One of the mechanisms is Automation and another is friendly parameter types.

In COM every interface is inherited from `IUnknown` that defines the basic COM mechanisms (e.g., reference count). Alternative superinterface candidate `IDispatch` (inherited from `IUnknown`) defines Automation mechanism, a protocol that makes it possible to determine the services (methods) of the interface (or its implementor) at runtime, thus obliviating the need for header files or other similar mechanisms.

By friendly parameter types we mean the use of language neutral COM constructs (like safearrays) instead of more limiting and obvious choices that map directly to the implementation language (C++ arrays).

Unfortunately TED API has utilized neither the Automation nor friendly parameter types, thus hindering the use of other languages besides C++, especially scripting languages.

4 Architecture

In order to be able to use PythonCOM with TED API, an additional layer (which we have named Tedious) is needed. Tedious layer is another COM server, and it acts as a conduit between TED package (in Python) and the original TED API. The layers are shown in Figure 2.

A basic guideline for the TEDious Scripting architecture has been simplicity. The demand for simplicity derives from the nature of the TED API. First of all, the API is huge (nearly three hundred COM interfaces and nearly two hundred classes) and not very well commented or documented. In order to be able to cope with the sheer mass of code the Tedious layer is kept quite thin and the most of the actual refactoring of the API is done in the TED package (in Python). Some of the architectural principles and solutions are shown in Figure 3. While the class names `Bob`, `Dot` and `Reboot` are artificial, the figure still highlights the core issues.

There is no apparent interface/class dichotomy in the Tedious layer. In other words, while in TED API a class can implement multiple interfaces and for each interface there does not have to be a class with the same name, the Tedious layer upholds one-to-one relationships between wrapper interfaces and wrapper classes. Thus in the example, the interfaces `IBob` and `IDot` have their wrapper interface/class counterparts in the Tedious layer and so has the class `CReboot`.

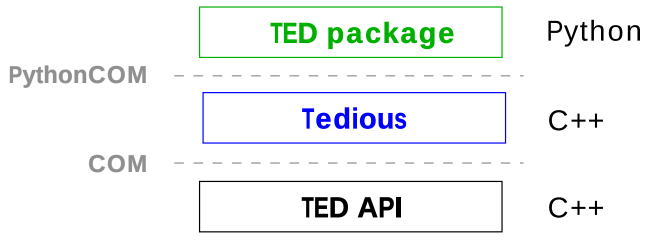

Figure 2. TEDious Scripting layers.

In the traditional COM way of doing things, a class can implement multiple interfaces and each interface can be seen as a different view of the COM object. In the Tedious layer, instead of using COM `QueryInterface` mechanism, the different views are represented as corresponding `Query` methods of the interface that is unique to each class. So the return value of `IWrapReboot::QueryBob` would be a `IWrapBob` instance that wraps the `CReboot` instance (viewed through `IBob` interface), whose wrapper's `QueryBob` method got called. As we see the end result semantics are quite identical to the regular COM mechanism. The main benefit from this approach is the simplicity: the wrapper interface tells everything we need to know about the class.

The class/interface issue is finally resolved and made invisible in the top TED package layer, where the lower Tedious layer interfaces and their relationships are modeled as a class hierarchy with properties. The property protocol for the TED package is defined by class `SetGet`, and so it is a base class for all property-aware classes. Each property has a name and properties are introduced via `properties` class attribute. A class supports both the properties it has itself declared and the properties it has inherited. The protocol interface consists of dictionary-like operators (indexing by name and `has_property`), `set` and `get` methods, and helpful method `property_names` that returns the names of every property the object supports.

The aggregation relationships between the different layers represent the actual wrapping. While the aggregation "hierarchy" seems pretty straightforward (`Bob` instance *has-a* `IWrapBob` instance *has-a* `IBob` instance) there are some subtle issues. One obvious issue is the level of abstraction. Whereas the Tedious layer bears a great resemblance to the original TED API, the TED package does not give nearly anything away about its roots. As a matter of fact, apart from the existence of the `Wrapper` class the developer does not even come across any wrapping mechanisms.

There are two issues in the wrapping problem space: the instantiation of the correct wrapper class, and its initialization. Each instance of any class defined in the Tedious layer (Tedious object or `tob` for short) returns its class name via `ClassName` method (eg., `IWrapUmlClass::ClassName` returns "UmlClass"). Thus, with the help of Python's `eval` function, the wrapping of a `tob` is pretty straight-forward. When the constructor of a wrapper class is called, it has to initialize all of its base classes. Fortunately Python's flexible object model makes it possible to implement this initialization procedure in the constructor of the `Wrapper` class, thus removing the need for other wrapper classes to have constructors at all.

Another main guideline in TEDious Scripting has been incremental development, since wrapping the whole UML metamodel taxonomy of the TED API at one go would not be practical, useful or perhaps even possible. The interfaces/classes to be wrapped are selected on the basis of necessity: for example UML class diagrams need `UmlClass` more urgently than `UmlNamespace`, and thus `UmlClass` should be implemented first of the two. However, according to IDL files the `UmlNamespace` should be a base class for `UmlClass` in `ted.uml`. How can one implement a subclass before its base class? The question becomes trivial when analyzed in the terms of the Tedious layer. In other words, it would mean that a wrapper for a class in the TED API would not support all the interfaces of the class. Incrementality would also mean that when a new interface (like `IUmlNamespace`) gets wrapped, the corresponding implementor class wrappers must be updated as well (like `IWrapUmlClass`), and the `ted.uml` hierarchy must reflect these changes. Fortunately much of the work is done by Visual C++ ATL Object Wizard and Python scripts.

5 Evaluation

Listing 1 shows TEDious Scripting in action. The example script inserts an extremely simple class diagram into the TED repository. The constructed diagram represents the Singleton design pattern from the so called Gang of Four book [5].

Like many UML design tools TED separates the actual model from its views. This separation is reflected in the repository structure by two parallel hierarchies: the

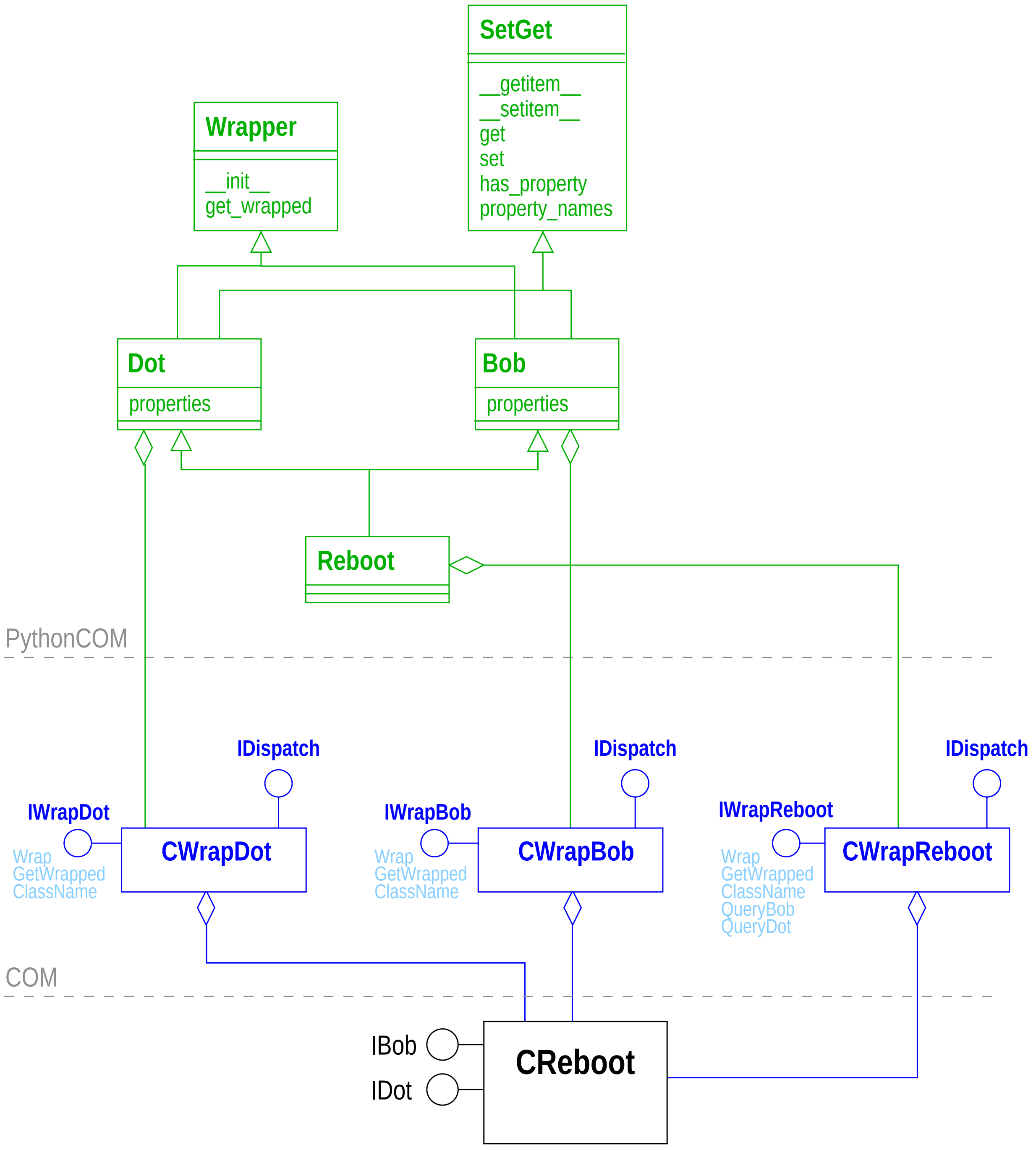

Figure 3. TEDious Scripting architecture principles.

model and the view. In our example we have chosen to model a UML package that contains `Singleton` class. The class has a view counterpart that is stored in a so called workbook.

Due to the simplicity of the diagram the code in Listing 1 is very straightforward. After the connection to the repository named `sandbox` that resides on host machine `hessu`, `UmlPackage` instance is created below the root node of the model hierarchy. Subsequently the rest of the Singleton model is created accordingly. On the view side the first artifact added to the hierarchy is the workbook (`DenPlace` instance). While the workbook is part of the view hierarchy it does not have a model counterpart as such. Thus the actual Singleton view is the `UmlVNode` instance that results from the `create_view` method call of the `UmlClass` in-

```python
import sys
from ted import *

try:
    rep = repository.Repository( 'hessu', 'sandbox' )
    # Model
    # GoF package
    pkg = rep.model_root().create_child( uml.UmlPackage )
    pkg['name'] = 'GoF'
    # Singleton class
    cls = pkg.create_child( uml.UmlClass )
    cls['name'] = 'Singleton'
    # operations
    cls.create_child( uml.UmlOperation )['name'] = 'static Instance'
    cls.create_child( uml.UmlOperation )['name'] = 'SingletonOperation'
    cls.create_child( uml.UmlOperation )['name'] = 'GetSingletonData'
    # attributes
    cls.create_child( uml.UmlAttribute )['name'] = 'static uniqueInstance'
    cls.create_child( uml.UmlAttribute )['name'] = 'singletonData'
    # View
    # GoF Workbook
    wb = rep.view_root().create_child( den.DenPlace )
    wb['label_text'] = 'GoF'
    # View for Singleton class
    view = cls.create_view( uml.UmlVNode, wb )
    view.set( location = mechanics.new_point( 10, 10 ),
              size = mechanics.new_size( 200, 150 ) )
    repository.synchronize()
    print 'Repository modified.'
except:
    sys.stderr.write( '%s: %s\n' % sys.exc_info()[:2] )
```

Listing 1. Repository modification.

stance. Besides the type of the view element the method needs to know the parent view element in order to connect the view properly into the view hierarchy.

For a reader, who is not familiar with TED API, it is probably quite difficult to evaluate the example. A rude method is to compare the example with its C++ counterpart that modifies the repository in the same way. The most notable difference between the two versions is the size. The C++ version is about five times longer, when measured in the lines of code. This is mostly due to different types of error handling. Whereas in C++ a programmer has to cope with `HRESULT`s, the PythonCOM uses Python's exception mechanisms. PythonCOM also lessens the programmer's burden by taking care of the most mundane tasks of the COM programming. These include conversions (Python strings $\leftrightarrow$ `BSTR`s, sequences $\leftrightarrow$ safearrays) and reference counting, among others. These features, combined with TEDious Scripting's way of wrapping the TED API as a class hierarchy, actually hide the COM origin of the library. In other words, TEDious Scripting integrates seamlessly with other Python modules.

One essential benefit gained from using Python is the possibility to access a repository interactively via Python interpreter session. This is useful not only for demonstration purposes, but for simple tasks like checking the status for an element. This kind of access is especially useful in those cases when we are interested in the properties not accessible via the TED tool and its graphical user interface, like unique `id` of an element.

Another issue is efficiency: one could easily think that two layers of wrappers combined with COM and Python would amount to unacceptably slow execution speed. Fortunately initial experiences suggest that this is not the case. That is probably due to two factors: simplicity of methods and the runtime layout of the distributed repository-based design environment. Former factor means that method calls that get passed through the different layers have usually a very limited set of functionality, namely to get or set a property. The biggest overhead in this sense comes from property and wrapping mechanisms but even those become negligible compared to the latter factor, that of the system runtime architecture. The TED API grants access to TED repository, namely a database that is (in a typical TED installation) located somewhere in the network. While TED has a cache mechanism in the form of a local copy of the repository, there is still so much overhead that any other factors pale in comparison. However, that does not imply that TEDious Scripting is slow no matter what, quite the contrary. The experiences with both standalone scripts and interactive Python interpreter sessions have led us to believe that the system is fast enough.

6 Conclusions

During the TEDious Scripting project the use of Python has proved to be a fortunate choice, especially for four separate, yet equally important reasons: dynamic object model, `eval`, usefulness as a code generator and finally the very good support for COM.

Together with `eval` the `Wrapper` and `SetGet` implementations provide a developer with sort of dynamic protocols for wrapping Tedious objects and calling their methods through properties mechanism. In other words, while the protocol implementations are somewhat TED specific, they still act as a proofs of concept that emphasize the benefits of Python's object model and `eval` dynamics.

The Python's role in the development process can be summarized with another tip, Tip 28: [4]

> LEARN A TEXT MANIPULATION LANGUAGE
> You spend a large part of each day working with text. Why not have the computer do some of it for you?

Using Python we have successfully manipulated IDL and C++ files, and generated HTML (from IDL and Python code) and Python (from IDL). At the same time we have witnessed how throwaway Python scripts have been able to evolve into extendable parts of the TEDious Scripting implementation toolbox.

Some of the most important factors in the TEDious Scripting implementation were not the issues we had to resolve, but the problems that were solved for us. In other words, PythonCOM frees the developer from the most mundane and literally tedious tasks of COM programming.

TEDious Scripting was in active use for more than a year within ATOS[3] research group in Tampere University of Technology and the researchers were quite happy with it both from the user's and developer's (who wraps additional parts of TED API) point of view [6].

[3] `http://practise.cs.tut.fi/atos/index.html`

References

[1] Python language website. `http://www.python.org`.

[2] Mark Hammond and Andy Robinson. *Python Programming on Win32*. O'Reilly & Associates, 2000.

[3] Johan Wikman. Evolution of a distributed repository-based architecture. In *Proceedings of NOSA'98*, 1998. `http://www.hk-r.se/fou/forskinfo.nsf`.

[4] Andrew Hunt and David Thomas. *The Pragmatic Programmer*. Addison-Wesley, 2000. `http://www.pragmaticprogrammer.com`.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns — Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. 395 pages.

[6] Johannes Koskinen, Jari Peltonen, Petri Selonen, Tarja Systä, and Kai Koskimies. Model processing tools in UML. In *Proceedings of ICSE 2001*, pages 819–820, May 2001.