Scripted Components

Dr. James A. Bednar

jbednar@inf.ed.ac.uk

http://homepages.inf.ed.ac.uk/jbednar

Scripted Components: Problem

(Cf. Reuse-Oriented Development; Sommerville 2004 Chapter 4, 18)

A longstanding goal of software developers has been to be able to build a large application by gluing together previously written reusable components.

Despite decades of work and successes in some areas, such as system libraries, this goal remains largely unfulfilled. Why?

Problems with Components

Reuse of components is hard! E.g.:

- Components rarely have matching interfaces
- Existing components are difficult to adapt to new applications
- Vicious cycle: without expectation of reuse, no incentive for making reusable components

Single-Language Assumption

Ousterhout 1998: Part of the reason for low levels of reuse has been the mistaken assumption that components should be created and used in a single language.

Underlying problem: languages best suited for creating components are worst suited for gluing them together, and vice versa.

Requirements for Building Components

To implement useful, high-performance primitives, you typically need:

- Speed
- Memory efficiency
- Bit-level access to underlying hardware and OS

Systems Languages

The requirements for building components are met by systems languages like C and C++.

Systems languages allow (and typically require) detailed control over program flow and memory allocation.

With such power available, strong typing (e.g. strict inheritance hierarchies) is necessary to prevent catastrophic errors.

Requirements for Gluing Components

To glue primitive components written by multiple independent developers into an application, you want:

- Weak or no typing, to allow different interfaces to connect
- A small number of high-level, widely shared interface datatypes (e.g. strings, objects)
- Automatic memory management, etc., to allow one-off data structures to be created easily for gluing
- Graceful user-relevant error handling, debugging

Scripting Languages

Gluing components from independent developers in systems languages requires huge amounts of code and much time debugging, often swamping any benefit of reuse.

Scripting languages excel at gluing, because they insulate the user from the details of program flow, memory allocation, and the operating system.

Scripting languages are good for manipulating (analyzing, testing, printing, converting, etc.) pre-defined objects and putting them together in new ways without having to worry much about the underlying implementation.

Scripting Language Features

Typically:

Interpreted: for rapid development and user modification

High-level: statements result in many machine instructions

Garbage-collected: to eliminate memory allocation code and errors

Untyped: to simplify gluing

Slow: for native code (but can use fast external components)

Examples: Python, Perl, Scheme/Lisp, Tcl, Visual Basic sh/bash/csh/tcsh

Scripted Components: Pattern

Use a scripted language interpreter to glue reusable components together, packaging an application as:

- An interpreter
- A component library (preferably mostly preexisting)
- Scripts to coordinate the components into a meaningful system

Applications can be tailored to specific tasks by modifying the script code, potentially by end users. Configuration options can be saved within the scripting language itself.

Scripted Components: Advantages

- Helps make maintaining a large code body practical
- Increases long-term maintainability because application can be reconfigured as needs change
- Promotes reuse
  (and thereby development of reusable components)
- Provides separation between high-level and low-level issues (and programmers?)
- Greatly reduces total size of code, and/or expands functionality

Scripted Components: Liabilities

- Can be complicated to bind languages together (but see SWIG)
- Must learn and maintain source code in multiple languages
- Can be slow if critical components are implemented in the script language
- Largest benefit requires existing components

Scripted Components: Example

Emacs editor, version 21.3

Core rarely-changed code written in C (265 KLOC), implementing custom LISP interpreter and performance-critical components.

Rest in LISP (580 KLOC), most of it user-contributed (i.e., written independently).

Maintained continuously for more than 30 years, by hundreds (thousands?) of people.

Scripted Components: Examples

Other examples:

- Matlab
- Gimp
- LaTeX
- Many domain-specific systems
- Anything with macros, a configuration file, etc.
  (all large programs?)

Custom vs. Off-the-Shelf

Most existing large, long-lived programs use custom languages for macros or configuration, but those are hard to maintain, hard to learn, not shared between programs (limiting reuse), and limited in functionality.

Modern approach: Plug-in scripting languages.
Many now available freely, with large bodies of reusable component libraries. Just download one and get to work!

E.g. Python, Guile (Scheme), Tcl, Perl

Java?

Where does Java fit into this worldview?

Ousterhout: Java is a systems language, good for implementing components, that happens to be interpreted (like a script language).

Me: Maybe. To me Java seems weak as a component implementation language (compared to performance of e.g. C++), but also weak as a gluing language (due to strong typing). Thus it seems like a compromise between scripting and systems languages, when Scripted Components offers best of both (at a cost of complexity).

Required Reading

Ousterhout 1998, http://home.pacbell.net/ouster/scripting.html

Note that the author developed the Tcl scripting language, and thus is strongly biased towards it. I personally think object-oriented scripting languages like Python are much better for scripting object-oriented components, because objects in the component language appear as native objects in the scripting language.

Nat Pryce, http://www.doc.ic.ac.uk/~np2/patterns/scripting/scripting.html

Summary

- Scripted Components pattern applies to many (most?) large-scale systems
- Allows high-level, abstract languages to be used for high-level tasks
- Allows low-level systems languages to be used for low-level tasks
- Provides and encourages component reuse
- Scripting languages now freely available
- Avoid writing custom configuration or macro languages

References

Ousterhout, J. K. (1998). Scripting: Higher level programming for the 21st century. *Computer*, *31* (3), 23–30.

Sommerville, I. (2004). *Software Engineering*. Reading, MA: Addison-Wesley, 7th edn.