Components, Scripts, and Glue

Jean-Guy Schneider

Software Composition Group
Institut für Informatik und angewandte Mathematik (IAM)
Universität Bern

E-mail: schneidr@iam.unibe.ch
WWW: http://www.iam.unibe.ch/~scg/

Overview

1. An Example
   - ☞ Identifying missing keywords

2. Scripting and Software Composition
   - ☞ Components, architectures, scripts and glue

3. Summing Up
   - ☞ Separation of concerns; some pointers

Part I: Example

Example: Extracting Keywords

Keyword master file:

```
Component models and definitions
```

Information files:

```
#Label: 3
#Mod: Thu May 21 11:50:34 MET DST 1998 schneidr
#Keys: Python Component
Components and Modules can be written in Python or in C/C++. A
client of a module is not aware whether it is written in Python
or in a system programming language. On systems with dynamic
loading, recompilation of the Python interpreter is not neces-
sary; a module itself has to be a shared library.
#see python:1
```

Problem:

- ☞ check for keywords missing from master file

First Approach: C Program

```
/* get keywords of a file */
List getKeys (char* fileName){
   FILE* file;
   char* line;
   char first[80];
   List newKeys = 0;

   file = fopen (fileName, "r");
   if (file) {
      while (!(feof (file))){
         line = read_line (file);
         if (*line != '#') {
   sscanf (line, "%s", first);
   appendToList (&newKeys, first,
         UNIQUE);
            free (line);
         }
      }
      fclose (file);
   }
   return newKeys;
}
```

```
/* Main */
int main (int argc, char* argv[]){
   List orgKeyWords = 0;
   List foundKeyWords = 0;
   int i;
   /* Get valid keywords */
   orgKeyWords = getKeys(KEYWORD_FILE);

   /* Loop over arguments */
   for (i=1; i < argc; i++) {
      appendListToList (&foundKeyWords,
   getKeyWords (argv[i]), UNIQUE);
   }

   /* remove keyword tag */
   removeFromList (&foundKeyWords,
      KEYS_TAG);

   /* display difference of lists */
   diffLists (orgKeyWords,
      foundKeyWords);
   return 0;
}
```

First Approach: Observations

- Approx. 100 lines C code (plus 200 lines library code)
- Compile-time type checking
- Non-trivial memory management (explicit `malloc` and `free`)
- User-defined data structures (e.g., lists)
- Complex control structures
- Difficult to adapt and extend
- Use of an object-oriented approach does not reduce code size considerably

Second Approach: Shell Script

```
#!/bin/sh
# Check for unknown keywords

awk '! /^#/ {print $1}' keywords | \          # get first word of non '#' lines
   sort > /tmp/$$                             # sort into temporary file

wrong="`grep -h '^#Keys' $* | \               # get lines with '#Keys' tag
   tr -c '[A-Z][a-z]' '[\012*]' | \           # split words into separate lines
   grep -v 'Keys' | \                         # remove lines with '#Keys'
   sort -u | \                                # sort, remove duplicates
   comm -13 /tmp/$$ - `"                      # compare with temporary file:
                                              # -13: contents unique to I-stream
if [ -n "$wrong" ] ; then                     # empty string in '$wrong'?
   echo "There are unknown keywords:"
   for i in $wrong ; do                       # iterate over unknow keywords
      grep -n "^#Keys:.*$i" *                 # display files and line numbers of
   done                                       # unknown keywords
else
   echo "All keywords are known"
fi

rm /tmp/$$                                    # remove temporary file
```

Second Approach: Observations

- 16 lines of source code
- Use of standard UNIX *components* (`awk`, `comm`, `grep`, `sort`, `tr`), text streams, and files
- Pipes and filters
- Simple expressions and control structures: simple *architecture*
- Regular expressions
- Automatic memory management (garbage collection)
- Extended functionality
- Extensible
- Run-time type checking

Part II: Scripting and Software Composition

Programming Paradigms

A programming language is a *problem-solving tool.*

Imperative style:

- program = algorithms + data

Functional style:

- program = functions ○ functions

Logic programming style:

- program = facts + rules

Object-oriented style:

- program = objects + messages

A Conceptual Framework for Composition

We can keep software systems open and flexible by building them out of components.

applications = components + scripts

Architectural style: formalizes standard component *interfaces*, *connectors*, and *composition rules*

Components: black-box entities *export and import services*

Scripts: specify a *composition* (i.e. an architecture)

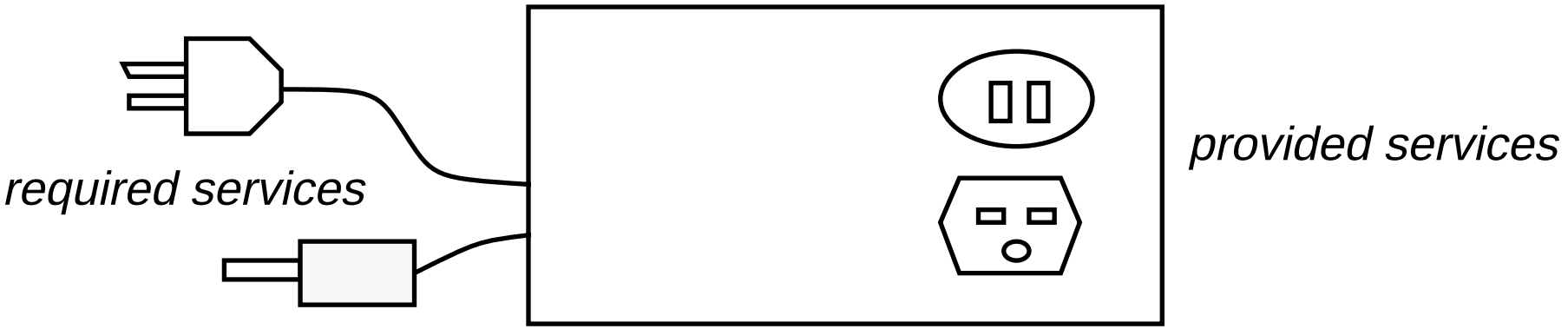
Coordination abstractions: implement the *connections*

Glue code: overcomes *compositional mismatches*

Components

Components are “designed to be composed”

- black box entities
- that provide services to other components
- and may also require services to work



“A software component is a composable element of a component framework”

Software Architectures

Software Architecture: describes a software system as a *configuration* of components and connectors.

Architectural Style:

- abstracts over a set of related software architectures
- defines a *vocabulary* of component and connector types and a set of rules governing their composition

Examples:

- Data flow: Pipes and Filters, Data-flow network, ...
- Independent components: Event systems, ...
- Data-centered: Repository, Blackboard, ...

What is Scripting?

Unlike mainstream component programming, scripts usually do not introduce new components, but simply wire existing ones.

— Clemens Szyperski

Scripting labels a high-level language that gets something outside itself (a browser, system facilities, ...) to do the work of an application.

— Cameron Larid

A scripting language is a high-level language to create, customize, and assemble components into a predefined software architecture.

Glue code

The purpose of *glue code* is to adapt foreign components that do not fit into the architectural style of a given framework.

```
# ad hoc glue code
source | filter > /tmp/in$$

foreign -i /tmp/in$$ -o /tmp/out$$    # not a filter!

cat /tmp/out$$ | finish

rm -f /tmp/in$$ /tmp/out$$
```

Glue Abstractions

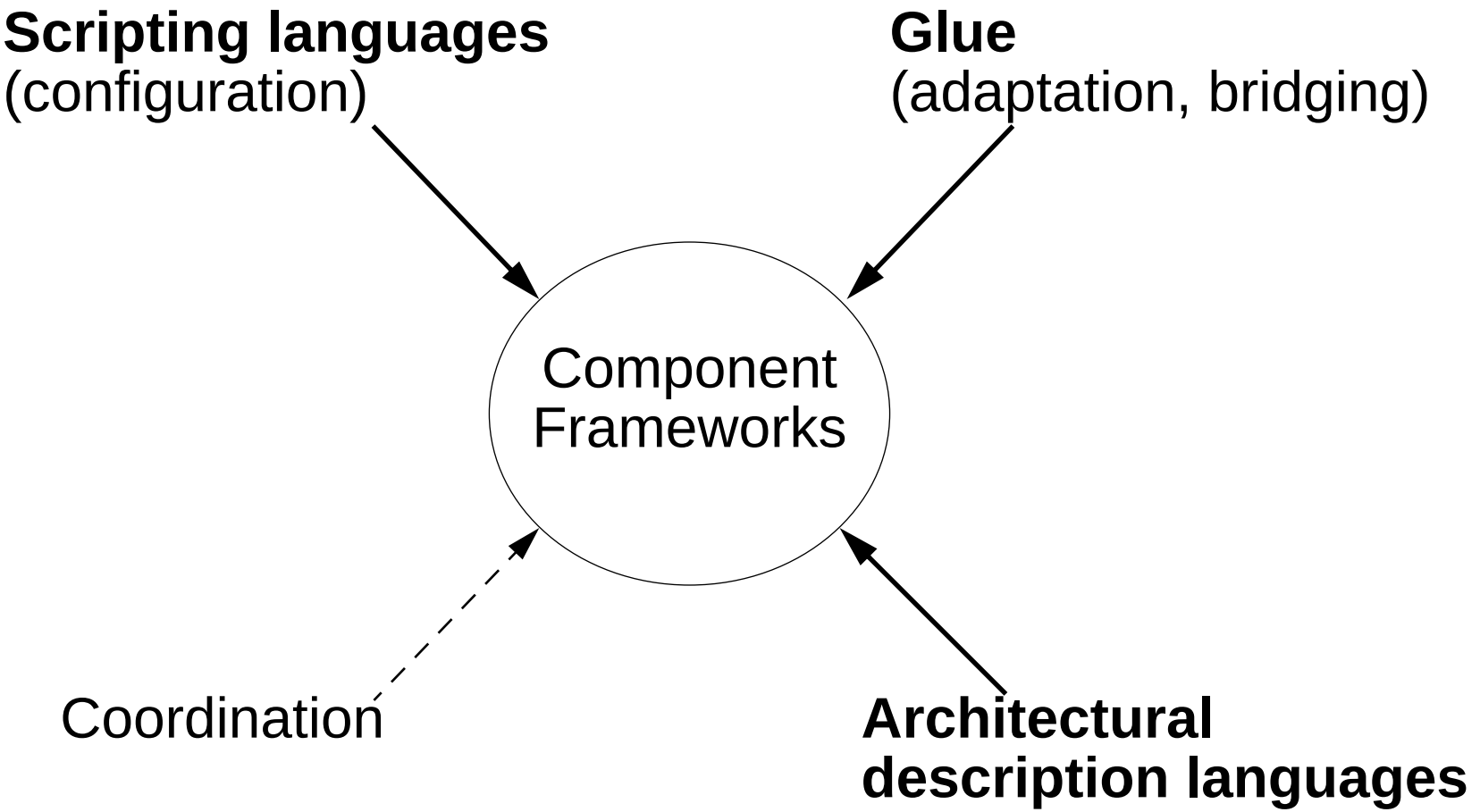
A *glue abstraction* defines a general way to bridge *compositional mismatch*:

```
# wrap -- a generic glue abstraction
foreign=$1
in=/tmp/in$$
out=/tmp/out$$
cat > $in
$foreign -i $in -o $out
cat $out
rm -f $in $out

# using the adapted component
source | filter | wrap foreign | finish
```

Part III: Summing up

Separation of Concerns



Separation of computational elements and their relationships

Scripting and Application Development

What I think is quite important, but underrated, is the dichotomy that scripting forces on application design. It encourages the development of reusable components (i.e., "bricks") in system programming languages and the assembly of these components with scripts (i.e., "mortar").

— Brent Welch

Scripting languages are used to create, customize, and assemble components into a predefined architecture.

Pointers to Further Information

For further information about scripting:

- Addendum to notes, errata,
- Complete source code of examples,
- Extended versions of packages, libraries,
- Pointers to languages (web-sites, references, etc.),
- Conferences,

www.iam.unibe.ch/~scg/Teaching/Tutorials/ECOOP99-Scripting/