Overview

1. Motivation:
   - Open systems development.

2. Applications = Components + Scripts:
   - A conceptual framework for software composition.

3. Scripting languages:
   - Instantiation and composition of components.

4. Summing up:
   - Research directions.
Open Systems are Families of Applications

Open systems undergo changing requirements:

An individual system may either be an *instance* of a generic *family* of applications, or a *snapshot in time* of a changing application.

Dortmund, January 30, 2001

Jean-Guy Schneider
Some Problems with OOP

Reuse comes too late:

- OOA and OOD are domain-driven, which leads to designs based on domain objects, not available components and standard architectures.

Overly rich interfaces:

- OOA and OOD are domain-driven, which leads to rich object interfaces, but component composition depends on adherence to restricted, plug-compatible interfaces.

Lack of explicit architecture:

- OO source code exposes the class hierarchy, not the object interactions.
- How the objects are plugged together is distributed amongst the objects.

Steep learning curve for applications and frameworks:

- Adapting an application to new requirements typically requires detailed study, even if the actual changes are minimal.

Need for a different approach to build open systems.
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”

---

Dortmund, January 30, 2001

Jean-Guy Schneider
Objects vs. Components

- **Objects** encapsulate services:
  - objects have identity, state and behaviour.

- **Components** are abstractions used to build object systems:
  - e.g., classes, templates, mix-ins, modules, ...
  - explicit composition interface ⇒ type-checkable,
  - in general, first-class, higher-order,
  - subsumes inheritance.
Component-based development is a popular buzzword, but why should we use components?

- main motivation: *reusability*
  - components avoid “reinventing the wheel”,
- closed world assumptions no longer valid:
  - support *independent evolution* of application parts,
- enhance *flexibility, adaptability*, and *maintainability* of software systems:
  - evolution by changing components or connections,
- building applications by plugging components together:
  - allow for a *higher-level application development*. 
A Conceptual Framework for Composition

Software composition requires a clear separation of computational elements and their relationships:

Applications = Components + Scripts

Components: black-box entities export and import services
Architectural style: formalizes standard component interfaces, connectors, and composition rules
Scripts: specify a composition (i.e. an architecture)
Coordination abstractions: implement the connections
Glue code: overcomes compositional mismatches
Software Architectures

A 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, ...
Scripting

A script specifies how components are instantiated and composed:

- separates components from configurations,
- scripts may be turned into components,
- some scripts implement composition abstractions (i.e. high-level connectors).

A scripting model requires:

- composition mechanisms (binding, wiring),
- encapsulation to turn scripts into components,
- (higher-order) abstractions to use scripts as connectors.

A script makes the architecture of a system explicit in the source code.

Dortmund, January 30, 2001

Jean-Guy Schneider
Coordination

Coordination is the management of dependencies between distributed components.

- typically concerns flow of information and scheduling of tasks,
- ideally coordination is separated from computation:
  - separation of scripts and components,
- use established coordination abstractions and architectures
  - since coordination is hard to get right [Carriero & Gelernter].

Examples:
- Bourne shell, Linda, Darwin, Manifold
Glue code 

overcomes compositional mismatches by adapting components to a given framework or architectural style.

Glue code adapts:

- interfaces,
- contracts,
- platform dependencies (binary representation etc.).

May be one-off adaptations, or may be generic glue abstractions.

Note the distinction between composition (wiring) and gluing (adaptation).
Adaptability and Extensibility

Applications evolve by changing the components or their connections.

Reconfiguring components:

- Adapt existing components (re-configuration of required services),
- Extend existing components (inheritance, composition),
- Add new components (compatible with old ones),
- Script new components (composition & abstraction),
- Adapt external components (glue),
- Adapt the script (re-configure the wiring).
**Scripting Languages**

Unlike mainstream component programming, scripts usually do not introduce new components, but simply wire existing ones. Scripts can be seen as introducing new behaviour but no new state.

Clemens Szyperski

Scripting labels a high-level language that gets something outside itself (a browser, system facilities, ...) to do the work of an application. Other metaphors that emphasize this role are "glue" and "bricks and mortar".

Cameron Larid

“A scripting language is a high-level language to create, customize, and assemble components into a predefined software architecture.”
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 necessary; a module itself has to be a shared library.

#see python:1

Problem:

check for keywords missing from master file

Dortmund, January 30, 2001
Jean-Guy Schneider
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++) {
    appendToList (&foundKeyWords, getKeyWords (argv[i]), UNIQUE);
  }

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

  /* display difference of lists */
  diffLists (orgKeyWords, foundKeyWords);

  return 0;
}
First Approach: Observations

- App. 100 lines of C code (add. 200 lines of library code),
- Compile-time type checking,
- Non-trivial memory management (explicit \texttt{malloc} and \texttt{free}),
- User-defined data structures (e.g., lists),
- Complex control structures,
- Difficult to adapt and extend,
- Use of OO approach does not reduce code size considerably.
Second Approach: Shell Script

#!/bin/sh
# Check for unknown keywords

awk '!' /^#/ {print $1}' keywords | \
  sort > /tmp/$$

wrong="`grep -h '^#Keys' $* | \
  tr -c '[A-Z][a-z]' '[\012*]' | \
  grep -v 'Keys' | \
  sort -u | \
  comm -13 /tmp/$$ - `"

if [ -n "$wrong" ] ; then
  echo "There are unknown keywords:"
  for i in $wrong ; do
    grep -n '^#Keys:.*$i' *
  done
else
  echo "All keywords are known"
fi

rm /tmp/$$

# get first word of non '#' lines
# get lines with '#Keys' tag
# split words into separate lines
# remove lines with '#Keys'
# sort, remove duplicates
# compare with temporary file:
# -13: contents unique to I-stream
# empty string in '$wrong'?
# iterate over unknown keywords
# display files and line numbers of
# unknown keywords
# 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),
- Extensible,
- Run-time type checking.
Characteristics of Scripting Languages

- Application development: plugging components together
  - primary focus: composition (not inheritance),
- Extensible:
  - add new components and connectors,
  - incorporating legacy code.
- Embeddable:
  - adapting behaviour of existing components,
- Making software architectures explicit:
  - explicit support for architectural styles,
- Favor high-level programming over execution speed,
- Dynamic typing (run-time type checking),
- Dynamic creation/execution of code...
Composition Language

- Scripting languages (configuration)
- Glue (adaptation, bridging)
- Architectural description languages
- Coordination

Separation of *computational elements* and their *relationships*. 

Dortmund, January 30, 2001
Jean-Guy Schneider
**PICCOLA - A Small Composition Language**

**Components:**
- Concurrent, distributed agents,
- Interfaces and contexts modelled by forms (extensible records).

**Architectural Description:**
- Algebraic view of Components and Connectors,
- Reasoning (properties and compositions...).

**Scripting:**
- Instantiation and composition,
- Scripts as components.

**Glue:**
- Gateways agents to foreign code,
- Wrappers, message interceptors.

**Coordination:**
- Agents communicate through (distributed) channels,
- Higher-order coordination abstractions.

Dortmund, January 30, 2001
Jean-Guy Schneider
Research Directions

1. Composition languages:
   - How to specify components, architectures, and frameworks?
   - How to specify applications as compositions?
   - How to reason about architectures and compositions?

2. Tools:
   - How to represent and manage framework knowledge?
   - How to visually present and manipulate software components?

3. Methodology:
   - How to map problem domains to component architectures?
   - How to drive application development from frameworks?
   - How to iteratively develop and evolve component frameworks and applications?
References