

# *Components, Scripts and Glue*

**Oscar Nierstrasz**

*Software Composition Group*  
Institut für Informatik (IAM)  
Universität Bern

oscar@iam.unibe.ch  
<http://www.iam.unibe.ch/~scg/>

# Overview

## I. A Conceptual Framework for Software Composition

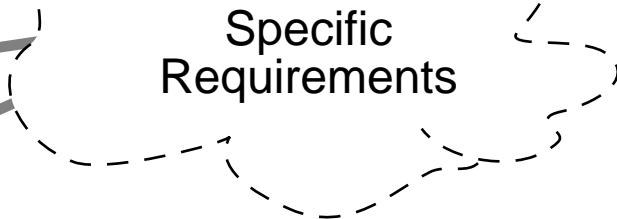
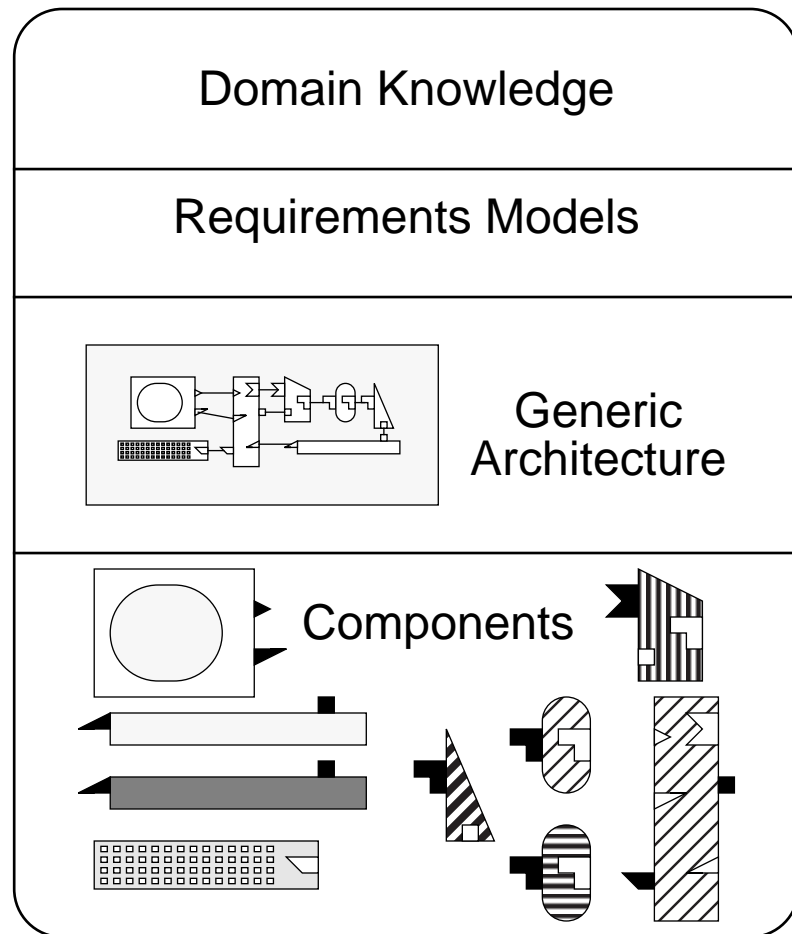
- ❑ Open Systems Development requires a Component-Based approach
- ❑ Adaptability through *explicit software architectures*
  - ☞ express applications in terms of *components, scripts* and *glue*
  - ☞ driven by standard *architectural styles*

## II. Research Activities

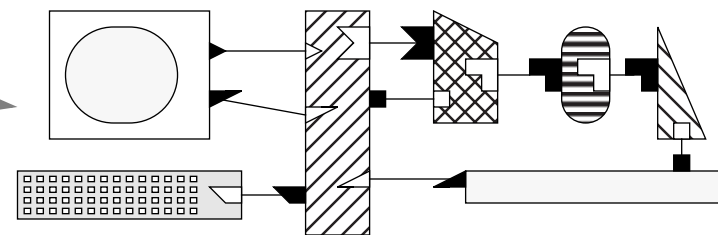
- ❑ Piccola — a small composition language
- ❑ FAMOOS — re-engineering OO applications to component frameworks

# Component-Oriented Development

A Component Framework



Component-Oriented Software Development is *Framework-Driven*



Specific Application

# *A Conceptual Framework for Composition*

*Ideally we want to build applications by “simply” plugging together components.*

## **Components**

- ❑ black-box entities that export *and* import services

## **Architectural style**

- ❑ formalizes standard component interfaces, connectors and composition rules

## **Scripts**

- ❑ specify a concrete composition (i.e., a concrete architecture)

## **Coordination abstractions**

- ❑ implement the connections

## **Glue code**

- ❑ adapts components that do not conform to the architectural style

# *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)

## *Some Problems with OOP*

*OOP is well-suited for component-based development, but current practice hinders it.*

### **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*

## Research Issues

### 1. Languages:

- ☞ How to specify components, architectures and frameworks?
- ☞ How to specify applications as compositions?

### 2. Tools:

- ☞ How to represent and manage framework knowledge?
- ☞ How to visually present and manipulate software components?

### 3. Methods:

- ☞ How to drive application development from frameworks?
- ☞ How to iteratively develop and evolve component frameworks?

### *Research Activities*

- ❑ Piccola — a small composition language
- ❑ FAMOOS — re-engineering OO applications to component frameworks  
+ other projects & experiments ...

# What is a Composition Language?

## **Scripting Languages**

*Configure applications from components*

*E.g., Perl, Python, Visual Basic*

## **Architectural Description Languages**

*Specify architectural styles in terms of components, connectors and composition rules*

*E.g., Wright, Rapide*

A Composition Language?



## **Coordination Languages**

*Configure applications from distributed, computational agents*

*E.g., Linda, Manifold*

## **Glue Languages**

*Adapt applications and components to new requirements and architectures*

*E.g., C, Smalltalk*

# *Piccola — A Small Composition Language*

## **Architectural Description**

- Algebraic view of Components and Connectors
- Reasoning (properties of compositions...)

## **Scripting**

- Extensibility: foreign code
- Extensibility: scripts as components
- Focus on wiring (not on programming)

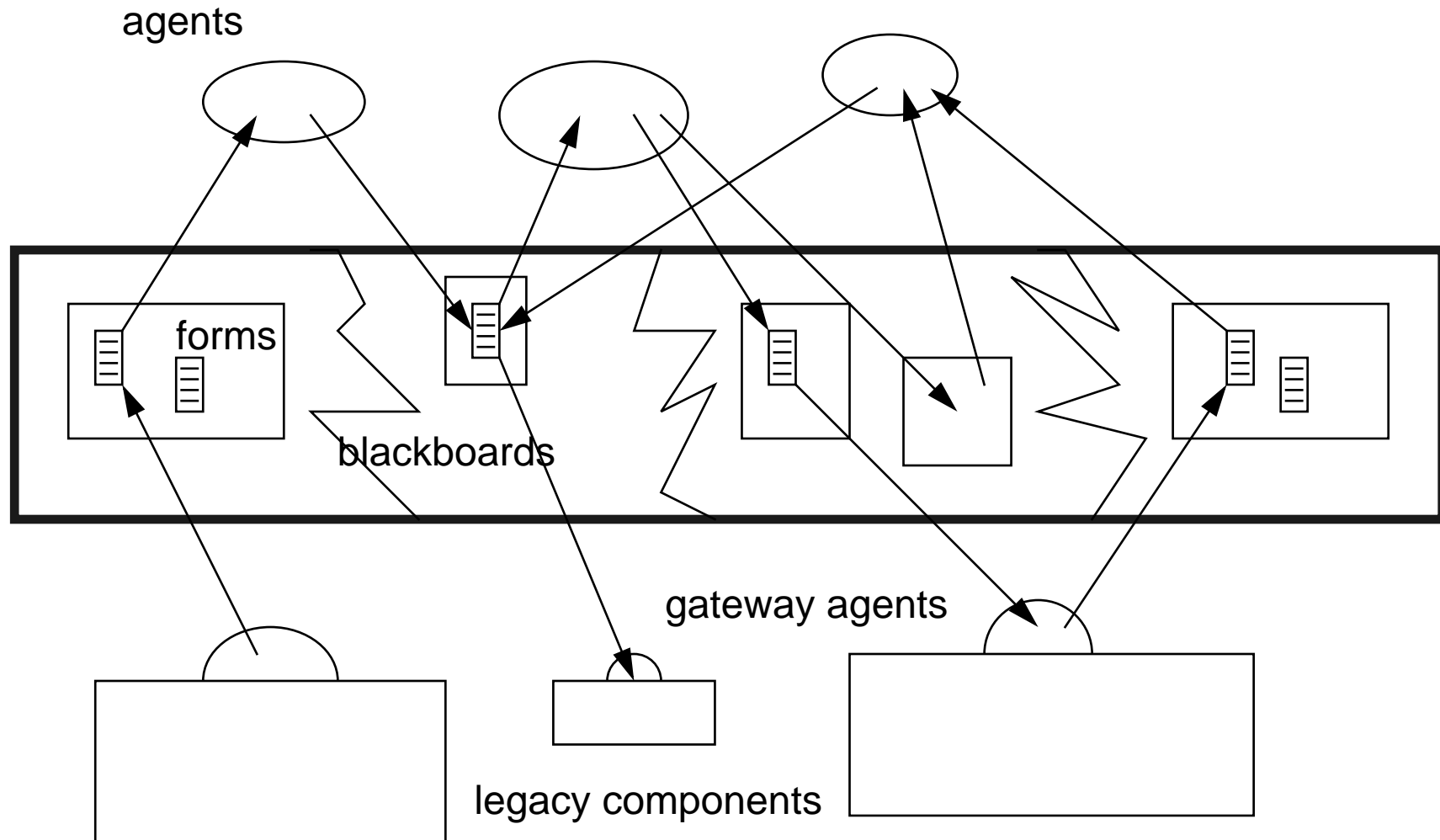
## **Coordination**

- Agents communicate through distributed blackboards
- Coordination abstractions

## **Glue**

- Gateway agents
- Adapters, message interceptors

# A Distributed Composition Medium



## *From pi to Piccola*

Add abstraction layers on the pi-calculus:  $pi \Rightarrow PiL \Rightarrow Piccola$

### **The asynchronous pi-calculus**

- ❑ reasoning about concurrency and communication, mature semantics

### **from pi to PiL**

- ❑ agents communicate *forms* (extensible records) instead of tuples
  - ☞ relax contract between sender and receiver
  - ☞ polymorphism
- ❑ conservative extension
- ❑ Java implementation of Agents, Forms, and Channels.

### **from PiL to Piccola Abstractions**

- ❑ Syntactic sugar for:
  - ☞ functions and procedures, early reply
  - ☞ nested forms, dynamic environment ...
- ❑ Built-in datatypes: Integer, Strings, ...
- ❑ Interface to Java via external channels. Java Objects as Forms.
- ❑ Persistent abstractions as (reusable) modules

## User-defined Abstractions

### Future:

```
export def future(service)(args) =  
  slot = global.newBlackboard()      # requires newBlackboard  
  return (val = slot.read)           # return access channel  
  slot.write(service(args))          # and write result
```

### Generic Wrapper:

```
export def wrap(service)(args) =  
  p = service.pre(args)              # invoke pre Wrapper  
  res = service.val(args)            # main service  
  return res                         # return its result  
  service.post(p)                   # and do some cleanup...
```

## Ongoing work ...

### **Foundations:**

- Type system for PiL
- Reflection (first class labels) for forms
- Reasoning on Composition

### **Language Design:**

- High-level syntax
- Abstraction mechanisms
- Supporting architectural styles

### **User Abstractions, Modelling:**

- Generic Synchronization Policies (self calls)
- Object/Component models

### **Pragmatics:**

- Applications
- Java Integration
- Visualization of agents and configuration

# FAMOOOS

*Re-engineering object-oriented legacy systems (!) towards component frameworks.*

## **Approach — apply a re-engineering life cycle:**

- ❑ Reverse engineering — model capture *tools* and *metrics*
- ❑ Problem detection — recognize “*legacy patterns*”
- ❑ Problem analysis — apply *re-engineering patterns*
- ❑ Change propagation — forward engineer

See: <http://www.iam.unibe.ch/~famoos/>

## *Observations:*

- ☞ components are not “just there for the taking”
- ☞ it’s hard to tell where the architecture is broken
- ☞ semantics-preserving transformations are not enough!

# Reengineering problems

## High-level problems

- insufficient documentation
- lack of modularity
- duplicated functionality
- improper layering

## Low-level problems

- misuse of inheritance
- missing inheritance
- misplaced operations
- violation of encapsulation
- missing encapsulation

## *FAMOOS tools & techniques*

### **Moose MetaModel**

- ❑ common meta-model and repository for reverse and re-engineering

### **Metrics**

- ❑ problem detection
- ❑ identifying refactorings between versions

### **Duploc**

- ❑ detecting and visualizing duplicated code

### **CodeScape**

- ❑ visualizing metrics to detect design problems

### **Reengineering Patterns**

- ❑ techniques to get from a legacy solution to a refactored solution
  - ☞ e.g., *Type Check Elimination in Clients*

### **Gaudi**

- ❑ extracting architectural artifacts from static and dynamic information

## Conclusions

*Components alone are not enough to build flexible systems.*

- ☞ We need component-based software architectures.

*OOP alone is also not enough.*

- ☞ Current practice conflicts with component-based development.

*CORBA, Delphi, COM++, Java Beans etc. etc. are also not enough!*

- ☞ Though useful, each tackles only specific technical issues.

*A conceptual framework for software composition ...*

- ☞ Think in terms of components, architectures, scripts, and glue.

*Research problems ...*

- ☞ Although there are interesting technical problems, the hard problems are methodological and cultural!