Picolo: A Simple Python Framework for Introducing Component Principles

Raphaël Marvie
LIFL – University of Lille 1 (France)
raphael.marvie@lifl.fr

Abstract

Components have now become a cornerstone of software engineering, and their use still increases. While the definition and implementation of a *component* may vary from one technology to another, the core concepts—unit of reuse, composition, and extensibility—are quite well identified. Developing and using components implies the understanding of these core concepts.

While a 10 years old child can discover basic electronics through experiments using an activity kit, learning components is not an easy task due to the complexity and overload of technical details that come with most general purpose component platforms. Moreover, component tutorials are most of the time dedicated to a particular technology API, not to the inner mechanisms.

Picolo is a minimal framework aimed at easing the introduction of components and component-based applications. Its purpose is to provide a simple support (a few hundred lines of Python code) for studying and understanding the core concepts and mechanisms related to components. Unlike general purpose platforms, it allows you to dive in and manipulate every aspect related to a component then to component-based applications. This paper presents and discusses its current design, implementation and use.

1 Introduction

Component Based Software Engineering (CBSE) [4] has become one of the leading approaches to developing software. Going beyond Object Oriented development, this approach promotes the shift from craft to industry for the field of software engineering.

This paper proposes a simple component model in order to allow the introduction of component technologies without the technical overload of most current component-based solutions. An introduction to components should not only answer *how to write a component in technology X* but it should explain the core mechanisms of components. Developing a component using technology X mainly focuses on how to use the API, not understanding the concepts. Thus, a technological switch may imply to learn the new technology from scratch as its API is certainly different from the first one. Understanding the core mechanisms of components permits you to more easily learn new technologies and their API.

Through a few hundreds lines of Python code, the *Picolo* framework allows you to dive into a component technology and to learn, study, modify, even extend its implementation. Before producing an application using components, it is possible to study the underlying mechanisms related to connecting components or sending a request to a component. This paper does not claim to present *the true approach* to components. It presents a point of view about components and their introduction based upon several years of research experiments, component platform developments and teaching. The *Picolo* framework is available under the GNU General Public License [7].

1.1 Background and Motivations

Promising the ease of using Lego for building applications and the reuseness of electronic pieces, software component technologies are more and more widely used. One of the main benefits advocated by component-based software engineering is to reduce the amount of technical code to be written. The environment hides some of the technical requirements of applications. However, it is not because the developer does not write parts of the applications that (s)he should not be aware of them and their behavior. Using something without understanding it generally leads to failure.

Definitions of the concept *component* exist and are commonly accepted [17], but its introduction is not an easy task. There are mainly two kinds of component environments: general purpose industrial ones and dedicated academic / research prototypes. General purpose component solutions—like the CORBA Component Model (CCM), the Enterprise Java Beans (EJB), the Fractal component model, or even the Web Services—are two heavy for allowing one to understand and manipulate the inner mechanisms of components. In the meantime, research component environments focus on particular aspects of components and do not necessarily cover the complete scope of CBSE.

While a 10 years old child can learn electronics through experiments using an Activity Kit, one cannot easily discover what is a software component and how it works. Even if open source component platforms are freely available and could be used for teaching purpose, their complexity reduces their usefulness as introduction material. The OpenCCM platform [8, 14] (Java implementation of the CORBA Component Model specification) is over 300K lines of code, the Jonas platform [13] (implementation of the Enterprise Java Beans specification) is about 32MB of source code, while Julia (Java implementation of the Fractal component model [12] that is only intended to be the core of a component environment) is about 20K lines of code.

Using those platform to introduce components may look like using a molecular accelerator for understanding that an H_2O molecule is made of two atoms of hydrogen and one atom of oxygen.

1.2 Proposal Overview

Unlike most component models, *Picolo* first intent is not to be useful nor suitable for building component-based applications. There are plenty of existing component solutions that already answer these topics. Even if they do not match all users' requirements, the goal of *Picolo* is definitely not to fill the gap.

The *Picolo* framework is intended to study *what is a component* (see section 3) and *how it can be implemented*. The framework allows you to experiment the underlying mechanisms of components such as how to create, connect and invoke operations on components. While using general purpose component model, you cannot easily understand the inner mechanisms. First, they are deeply hidden behind API and tools. Second, if you want to have a look behind the API, the amount of technical information to understand is overwhelming. Moreover, tutorials for component technologies are mainly focusing on the API and tools, not the concepts.

In order to reach this goal, the framework provides a set of classes that implements the core concepts and mechanisms of component environments (see section 4). Each concept and mechanism is thus reified and can be used to set up experiments in order to understand their meaning (see section 5). Some modules are also provided to present how extensions can be defined on top of the core mechanisms (see section 6). Finally, the framework also includes the definition of basic services common to component platforms, for example to support component life-cycles.

A last difference between *Picolo* and others component platforms is that the implementation of components follow a prototypical approach. It is not required to define component 'classes' before creating instances of components. This approach is similar to the *Self* programming language [18, 16]. Extending Python interactive sessions, a component can be created incrementally, putting the pieces together and looking at the result. Such an exercise is interesting for the learner to discover what is usually performed by the component platform and completely hidden to the user.

This framework is expected to be useful as an introduction to component base software engineering, before dealing with industrial component solutions like Enterprise Java Beans, the CORBA Component Model, Fractal, or any other alternative.

2 General Principles of Components

2.1 Characterizing Components

In the context of this paper, we consider a component to be a piece of software with explicit connection points named *ports*. A *port* is either provided by a component to its users, it is then an *input port*, or used by a component to access other components, it is then an *output port*. Ports provide access to sets of operations (that are similar to object methods). Operations are provided by input ports and used through output ones.

Using the traditional vending machine example, the vending machine itself is defined as a component. It provides several input ports: one for customers to buy a drink, one for providers to reload the machine with drinks, and one for repairmen to repair the machine. It also uses several output ports: one to be connected to a power supply and one to be connected to a water supply. Finally, the input port for customers offers three operations: to select a drink, to pay the drink, and to take the drink. The output ports do not provide operations, it is not the power plug but the power supply that delivers power.

Comparing software components and electronic ones is quite common as

we would like software components to be like electronic ones: building an application like we build a computer by assembling pieces. Figure 1 presents a parallel between electronic and software components discussed in this paper. An electronic component is defined with leads that receive (input) or emit (output) electric signals. Each lead is intended to allow the connection between electronic components. A software component is defined with ports (similar to leads) that receive (input) or emit (output) requests. Each port is expected to be connected to other ports of software components.

As a graphical convention, we are representing input ports on the left side and output ports on the right side of components. Output ports are represented by half-circles, they are simple "plugs" and do not really contain processing. Input ports are represented as circles, they contain processing (the operations), the behavior of the component.

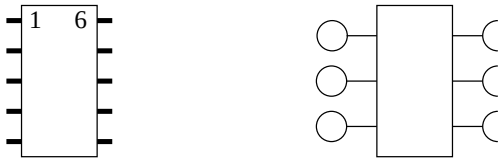

Figure 1: Comparison of Hardware and Software Components

Components are expected to be connected (composed) in order to build *assemblies*. An *assembly* represents a part of or a whole application. Composition is intended to be the basis activity of component-based software engineering: Components are bought off the shelf and composed to create application. It reduces the need for technical skills, and an domain expert can build an application using components. In the meantime, we will always need developers to create the components and put them on the shelf.

In the default version of the framework, an output port is connected to at most one input port, and input ports can be used by several output ports[1]. Once assembled, component default interaction mode is synchronous: Operation invocation is blocking (the caller instance wait until the called one has answered its request).

Figure 2 presents an interconnection of four software components. The component `customer` is connected to a component `vending machine`, meaning `customer` can use the services provided by `vending machine` through the port they share. Then, the component `vending machine` is connected to the components `power supply` and `water supply`, meaning it can use the services they provides through the ports they share. This configuration suggests that when `customer` sends a request to `vending machine`, the latter will probably send request to `power supply` and / or `water supply` for processing the request received from `customer`.

[1]The framework is not multi-threaded, thus invocations are received sequentially. However, multi-threading could be implemented as an extension.

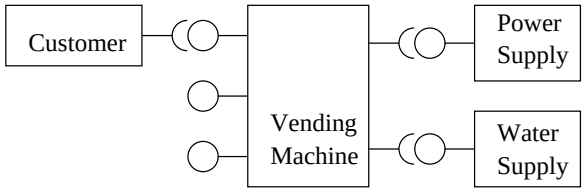

Figure 2: An Assembly of Software Components: The Vending Machine

2.2 The Meanings of *Component*

The term *component* is often used to designate different aspects of components such as a *component type*, a *component implementation*, a *component package*, or a *component instance* [10].

A component type is an abstract definition of a piece of software specifying its ports (input and output ones) as well as its properties (potentially the interaction mode). Being defined independently of implementations, the type can be implemented in various ways and thus allowing the substitution of *component instances* actually used when running applications.

A component implementation groups both the functional (business logic of the component) and non functional implementations (technical elements such as persistence or security) of a component. In an ideal way, both implementations are developed by distinct means and composed. The development of the non functional part should be automatically generated from a description (see section 7).

A component package is a deployment unit, often an archive, containing all the elements necessary to create an instance of the component on a particular host: at least the component type definition, an implementation, and a description of the archive content. This unit is expected to be available off the shelf in a *binary form*[2].

A component instance is, like an object, a execution entity living in a system. A component instance is defined by a unique reference, and is associated to a single component type and component implementation.

3 Definition of a *Picolo* Component

3.1 Overview

Picolo components are structured into three levels, each one dedicated to a specific concern.

Unit A component is intended to define a well identified software unit. This is true for component types, component instances as well as component implementations. Regarding a component instance, this means that the instance can be handled as a whole for creation, destruction, identification,

[2]This aspect of components will not be further discussed in this paper.

and so on. This handling is performed without having to care for each part of the component. For example, the vending machine is built of several pieces but is considered as one software unit.

Connection A port is intended to define connections between components instances in order to allow their cooperation. We are not looking at the whole unit any more, but at a points of interactions: A set of operations provided for a particular kind of clients. For example, the port of the vending machine used by a customer component instance provides a view on the interactions possible between a customer and the vending machine.

Behavior and state Operations and properties respectively reflect the behavior and the state of a component instance. At this level, we look at specific interactions between two component instances connected through two ports. For example, the customer invokes the operation *select a drink* on the proper port of the vending machine.

3.2 Structural Definition

In the same way as Python objects are defined as a set of members, *Picolo* components are defined as a set of *ports* and a set of *properties*. Ports and properties may be dynamically added or removed from a component definition.

Properties represent the "state" of a component instance, *i.e.* its inner structure and associated data items. They can be compared to object attributes, but speaking of properties imply the use of accessors (getters and setters) to handle the property values. Any data item related to the state of the component has to be defined as a property. The main argument is that the state of the component instance has to be shared between all its elements: Properties are available to the various operations of the various ports of a component instance.

Ports are connection points and may be of two kinds: input and output ones.

- *Input ports* provide a set of *operations* to clients of a component instance. An operation is defined as a name and a list of parameters. It can interact with the state of the component instance. Such a port represents (at least) part of the component behavior (as being part of its functional implementation). A port only contains functional elements of a component and no elements related to its state (that are defined as properties).
- *Output ports* provide connection points, in order for a component instance to interact with other component instances. They are intended to be used by the implementations of operations attached to the same component as the output port. Output ports provide access to the operations defined in the input port of another component connected to the output port: The set of operations available from the output port, is the set of operations provided by the connected input one.

3.3 Invocation and Lookup

Based upon the three levels of a component structure we can add the following about lookups. There is two kinds of lookups, each one being performed at different levels: a lookup related to component connections and a lookup related to operation invocation.

Components only share a partial view of each other, as two components are connected through a pair of ports. A first 'lookup' has to be performed in order to select the proper ports to be connected. This lookup is performed when the application is set up, and is performed at the component level (the unit). The component on which a port is requested looks at the list of ports it knows in order to return the proper port reference or raise an exception.

The second kind of lookup is performed when an operation is invoked, it is computed at the port level (a particular connection). The input port on which the invocation is performed looks at the list of operations it knows in order to make it happen or to raise an exception. An output port on which an operation is invoked does not perform any lookup: it simply forwards the invocation to the input port to which it is connected.

While the operation lookup is similar to the method lookup in object technologies, it is a single level one: the port knows the operation or not. There is no multi-level lookup as in the context of object hierarchies for supporting inheritance.

3.4 Separation of Concerns

This first look at components underlines the fact that a component is itself a composition of various elements: Each element is part of its state or of its behavior. Second, components follows the separation of concerns principle [6, 15]: First, various view of the component can be defined through its input ports, and second, its state and behavior (functionalities) are not mixed up. Finally, a component dependencies (related to other components) are explicitly defines through output ports. This last remark is sometimes underlined as a limitation of object technologies where functional dependencies are not explicit at the definition level but hidden inside implementations.

Figure 3 depicts the *Picolo* component model definition. A component is a set of ports, which may be input or output ones, and a set of properties. An input port is defined as a set of operations (which are not detailed here, as they are similar to object methods). Together with components, the *Picolo* framework definition includes connections. A connection is defined between an output port and an input one, and can be established and removed. The connection in itself is just the wire between components.

4 Implementation of a *Picolo* Component

For simplicity and flexibility reasons, the *Picolo* framework has been implemented using Python. The complete base implementation is about 300 lines of documented code, which makes it quite easy to understand[3].

[3]The code could be much shorter, but from a software engineering point of view it is better to use protected attributes with associated methods than public attributes.

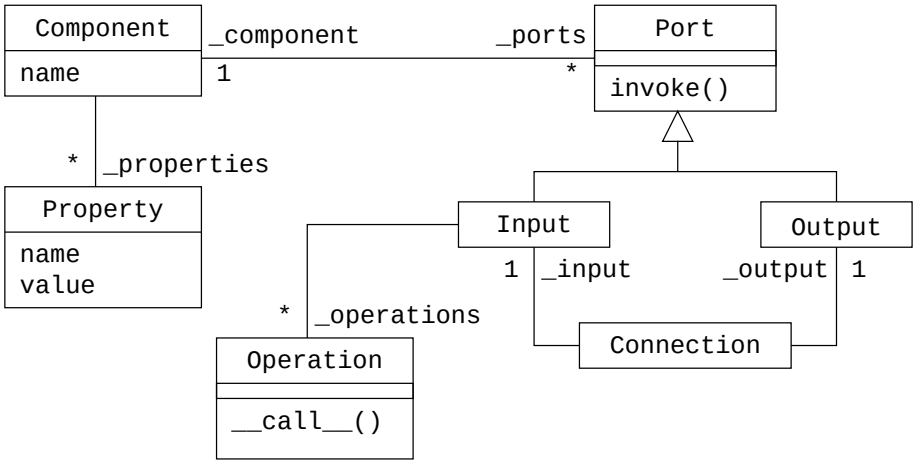

Figure 3: Definition of the *Picolo* Component Model

4.1 The *Picolo* Framework

The basic artifacts of *Picolo* components are objects: We have chosen an object oriented approach for its implementation[4]. An interesting benefit resulting from this choice is the ability to reify elements of a component: Every part of a component exists in memory as an object. This permits their handling in a dynamic way. While the ease of use is not a primary goal, it can be achieved through code generation from component descriptions: Only the functional part of the implementation has to be hand written (see section 7). The classes presented in this section are defined in the `core` module of the `picolo` package.

4.1.1 Components

The `Component` class is the heart of the framework. It manages the sets of properties and ports and provides associated methods. Associated with each set, two methods are defined for management purpose and two methods are defined for access purpose. Figure 4 presents only the methods related to port management (methods related to property management are similar, while accessors are trivial).

It is possible to dynamically associate a port (`bind_port`) or a property (`set_property`) to a component instance. A port or property instance is provided with a name, that has to be unique regarding the component instance (two ports may not have the same name). Similarly, it is possible to dynamically dissociate a port (`unbind_port`) or a property (`del_property`) from a component using their name.

At any time, it is possible to request from a component instance the list of its currently bound ports (`get_ports`) or its current properties (`get_properties`). The component returns a map which keys are the names of the ports or properties and values are the reference of the port or property instance. These two operations are useful as they provide support for the introspection of components: Given a component instance you can dynamically discover its ports and

[4] A functional implementation is also feasible.

```
class Component:
    def __init__ (self, name, ports=None, properties=None):
        self._name = name
        if not ports:
            ports = dict ()
        self._ports = ports
        if not properties:
            properties = dict ()
        self._properties = properties
    def bind_port (self, name, ref):
        self._ports [name] = ref
        ref._component = self
    def unbind_port (self, name):
        if name not in self._ports:
            raise UnknownPort, name
        self._ports [name] ._component = None
        del self._ports [name]
```

Figure 4: Base Implementation of a Component (excerpt)

properties. It is also possible to request a specific port (`get_port`) or property (`get_property`) given its name.

4.1.2 Input Ports

The `Input` class both manages the operations associated to the port and permits their use (see figure 5). The set of operations associated with an input port is managed as a map where names of operations are used as keys and the associated values are references to the object implementing the operation. This set can be provided at port instantiation and/or managed at runtime (using the `set_operation` and `del_operation` methods not presented here).

```
class Input (Port):
    def __init__ (self, ops=None):
        Port.__init__ (self)
        if not ops:
            ops = dict ()
        self._operations = ops
    def get_operations (self):
        return self._operations
    def invoke (self, opname, args):
        if not opname in self._operations:
            raise UknownOperation, opname
        if not self._component:
            raise UnboundPort
        op = self._operations [opname]
        return op(self._component, args)
```

Figure 5: Basic Implementation of an Input Port (excerpt)

You can introspect an input port using the `get_operations` method. It

returns a map of operations offered by the port. Together with the `get_ports` method provided by the `Component` class, this offers the complete structural introspection of a component instance.

The `invoke` method defines the mechanism to call an operation provided by an input port bound to a component instance. An operations is invoked providing its name (`opname`) and a sequence of arguments (`args`). The operation checks first if the operation is known to the port, and if the port is bound to a component (if not an exception is raised). Finally, the proper operation is evaluated providing two arguments: the component reference (to access its state) and the sequence of parameters.

4.1.3 Output Ports

The `Output` class manages the reference to an input port provided by another component (see figure 6). The synopsis of its `invoke` method is similar to the `invoke` provided by input ports. The implementation of the `invoke` method follows the *Proxy* design pattern [3]. Actually, the method only checks if the port is connected, raises an exception if not, and call the `invoke` method provided by the connected input port. Then, the availability of the operation is checked by the input port. The updating of the reference is performed by a `Connection` object.

```
class Output (Port):
    def __init__ (self):
        Port.__init__ (self)
        self._ref = None
    def invoke (self, opname, args):
        if not self._ref:
            raise PortNotConnected
        return self._ref.invoke(opname, args)
```

Figure 6: Basic Implementation of an Output Port

4.1.4 Operations

The `Operation` class is the base class for defining operations. Operations are defined as Python callable objects (meaning, they provide a `__call__` method). The behavior of an operation is defined (at least) as the `__call__` method. A simple function definition could also be used, however we have chosen a full object oriented approach to implement *Picolo*. This method expects two parameters: a reference to the component to which the port containing this operation is bound to, and a sequence of arguments[5] to be used by the operation. The first parameter is required to access the state and output ports of the component instance.

4.1.5 Properties

Any object can be used as the value of a property. As any class can be used to implement properties, no artifact is provided. As an example, if the state of

[5]It may be a Python list or tuple.

```
class Operation:
    def __init__ (self):
        pass
    def __call__ (self, cmp, args):
        pass
```

Figure 7: Base Implementation of an Operation

a component includes a counter, then an instance of any class implementing a counter can be used. It is important for a property which value can vary to use a mutable object, while non-mutable objects may be used to represent properties that should be read but not modified by the users. This does not really mind when the `set_property` method is used to erase the previous definition (meaning value) of the property.

4.1.6 Connections

The class `Connection` reifies a connection between an output port and an input port of two component instances (see figure 6). While port references are provided at instantiation of the connection (meaning at definition time), establishing (`connect`) and suppressing (`disconnect`) the connection can be performed dynamically. The `connect` method checks that the types of the ports provided are correct before connecting the two component instances (otherwise an exception is raised).

```
class Connection:
    def __init__ (self, oport, iport):
        self._input = iport
        self._output = oport
    def connect (self):
        if not self._input or not self._output:
            raise ConnectionNotConfigured
        if not isinstance(self._input, Input) or \
               not isinstance(self._output, Output):
            raise ImproperConnection
        self._output._ref = self._input
    def disconnect (self):
        self._output._ref = None
```

Figure 8: Implementation of a Connection

4.1.7 What About the Others?

Except for the availability of connection objects, the definition of a *Picolo* component is quite similar to a CORBA Component or a Fractal one[6]. While not so common in practice, the use of multiple explicitly defined input and output

[6]The author contributed to the finalization of the CCM specification, to the development of a CCM platform[9], and developed a simple implementation of the Fractal specification.

ports, or even multiple interfaces for a software element, was already existing in the Open Distributed Processing approach [5].

Component models such as the EJBs or the web services provide a single interface to their users. This single interface can be seen as a single port regarding this paper definition. In addition, they do not include the definition of output ports: The services used by a component (or a service) is not explicitly specified but hard coded in the component implementation. Finally, most other components models simply define components as *pieces of software* such as a library with entry points.

4.2 Discussing the Framework

4.2.1 Input Ports as Component Views

Input ports are views (or facets) of the component functionalities. Even if operations directly interact with the component state, they can be logically grouped together according to the various users of a component. It is common to see a vending machine for cold drinks as offering three views: one for customers, one for drink providers, and one for repairmen. Each view provides a dedicated set of operations according to the actor that uses the component. Moreover, non functional properties, like security, may vary from one port to another: While every one can buy a drink, only repairmen should open the back door of the machine.

4.2.2 Output Port Definitions

Output ports are not defined together with a set of operations. They implicitly provide the operations of the input port that is connected to them. This choice has been lead by the wish to support dynamic definition and manipulation of component instances: A component instance can entirely be defined interactively. This choice also fits the Python and Smalltalk *spirit*: send a message to an object, it will tell you if the message has a meaning or not.

The need for controlling the conformance between two ports (does the input port provides what the output port requires) is not considered as a requirement of the implementation. Such control can be achieved at a description level: Using a component definition language, you can check the conformance between two ports definition. The component implementation is then generated using this description (see section 7).

4.2.3 Non Functional Requirements

A non functional requirement of a component instance is any part of its implementation that is not directly related to its business logic. Persistence, transactions, security, and trace are common non functional requirements[7]. To attach a non functional requirement to an input port, you can simply extends the base `Input` class to implement the requirement specific processing[8]. This can also

[7] The qualifier *non functional* is used in this paper from the application level. A provider of a transaction service sees transactions as business logic.

[8] If the study of non functional requirements, and their support, is of particular interest for you, the `extension` module provides a framework for defining extensions. Moreover, the AOP (Aspect Oriented Programming) [2] approach can be implemented easily.

be applied at the operation level in order to attach non functional requirements only to particular operations.

The same approach can be applied to output ports, in order to perform processing related to non functional requirements before and / or after invocations on operations provided by the input port connected. Output port extension can be used to support distributed processing. Two component instances are running on two hosts, and they have to be connected together. The output port hides the fact that the invocation is sent through the network.

5 Example of *Picolo* Components

This section presents an example of component-based application using the *Picolo* framework we have just discussed. This application is a kind of "hello world" for components.

5.1 Defining Components

Using a free textual syntax, figure 9 presents the definition of two component types. The first one, `C1`, provides a single input port `input1` which offers the operation `foo`, and a single output port `output2`. The second one, `C2`, provides a single input port `input3` which offers the operation `hello`. These components are expected to be used together[9], the operation `foo` using the operation `hello` through a connection between `output2` and `input3`.

```
component C1
  input input1
    operation foo is Foo
  output output2

component C2
  property count is Counter
  input input3
    operation hello is Hello
```

Figure 9: Example of Component Definitions

In order to define components, you have first to implement the various operations and properties. Then, you can create component instances based upon the operations and properties you defined, and using the base classes of the framework (for defining ports and components).

5.1.1 Implementing Operations

The implementation of the `hello` operation follows the definition of callable objects in Python[10]. The functional code is contained in the `__call__` method. Figure 10 depicts this implementation. First, we retrieve the current value of

[9]Such information is not explicitly defined here, but some research works on Architecture Definition Languages have focused on expressing such dependencies in order to prove the well-formness of component assemblies[1].

[10]The `picolo.core` module is imported as `picolo` in all the paper, to ease the reading.

the `count` property of the component. The first parameter of the `__call__` method is a reference to the component that is used to access its state. Second, we increase the value of the property `count`. Finally, we return a string built up with the name of the caller passed as first argument and saying that (s)he is the *nth* caller of the operation (*n* being the value of the counter).

```
class Hello (picolo.Operation):
    def __init__ (self):
        picolo.Operation.__init__ (self)
    def __call__ (self, cmp, args):
        count = cmp.get_property('count')
        count.increase()
        return 'hello %s you are caller #%s' %\
            (args [0], count.value())
```

Figure 10: Implementation of the `Hello` Operation

Figure 11 presents the implementation of the `foo` operation provided by the component `component1`, which is similar to the implementation of the operation `hello` depicted in figure 10. The main difference lays in the fact that this operation uses an operation (provided by another component) through an output port. To do so, we first retrieve a reference to the output port named `output2`, using the reference of the component to which the port containing the operation is bound, and then we invoke the operation `hello` using the operation parameter sequence (containing the name of the user). Finally, we print the result of this invocation on the standard output.

```
class Foo (picolo.Operation):
    def __init__ (self):
        picolo.Operation.__init__ (self)
    def __call__ (self, cmp, args):
        ref = cmp.get_port ('output2')
        print ref.invoke ('hello', args)
```

Figure 11: Implementation of the `Foo` Operation

5.1.2 Creating Component Instances

Creating a component instance means first creating its internal structure, then adding its properties and ports. Figure 12 depicts the creation of an instance of component conforming to the definition of `C2` (see figure 9). First, we create an instance of `Component` providing the `component2` as the instance name. Second, we add the property `count` to the component instance. The property `count` is defined as an instance of the `Counter` class that is not presented here as being no more than a classic Python class implementing a counter.

Then, an instance of input port is created providing a map containing the unique operation to be offered through this port. The operation named `hello` is associated to an instance of the class `Hello` presented in figure 10. Finally, we

```
>>> c2 = picolo.Component ('component2')
>>> c2.set_property('count', Counter ())
>>> p3 = picolo.Input (ops={'hello': Hello ()})
>>> c2.bind_port('input3', p3)
```

Figure 12: Interactive Creation of the `component2` Component Instance

bind the input port just defined to the component `component2` with the name `input3`.

Finally, figure 13 depicts the creation of an instance of component conforming to the definition of `C1` (see figure 9). This creation is very similar to the creation of component `component2`, and only differs in the number of ports bound to the component and in the name of all its parts. We now have two component instances ready to be connected together.

```
>>> c1 = picolo.Component ('component1')
>>> p1 = picolo.Input (ops={'foo': Foo ()})
>>> c1.bind_port('input1', p1)
>>> p2 = picolo.Output ()
>>> c1.bind_port('output2', p2)
```

Figure 13: Interactive Creation of the `component1` Component Instance

5.2 Using Components

Most of the time, component instances are not used as stand-alone entities. Now that we have created two instances of components it is possible to connect them. Figure 14 presents, using a free textual syntax, the assembly of components we are going to create. This description first states that we are using the two instances created in the previous section, and second it specifies a connection between the output port `output2` of the component instance `component1` and the input port `input3` of the component instance `component2`.

```
assembly test1
  components
    instance component1 C1
    instance component2 C2
  connection
    from component1 through output2
    to component2 through input3
```

Figure 14: Example of Component Definitions

5.2.1 Connecting Component Instances

After defining a connection between the two ports, we can connect the two component instances (see figure 15). In order to connect two component instances,

you have first to define a connection: You create an instance of the `Connection` class using the two ports to be connected as parameters. Then, you actually establish this connection using the `connect` method. Before this invocation, the component instances are not connected together.

```
>>> cnx = picolo.Connection (p2, p3)
>>> cnx.connect()
```

Figure 15: Connecting Component Instances

5.2.2 Invoking Operations of a Component

Once connected, the two component instances become an application (a very simple one, but an application). We can then invoke the operation `foo` provided by the port `input1` of the component instance `component1` (see figure 16). The argument of the invocation is provided as a one element sequence (a Python tuple). When we invoke the `foo` operation, it makes the implementation of the operation invoking the `hello` operation. When `hello` returns its result, `foo` prints it on the on the standard output.

```
>>> ref = c1.get_port('input1')
>>> ref.invoke('foo',('Raphael',))
hello Raphael you are caller #1
>>> ref.invoke('foo',('Marvie',))
hello Marvie you are caller #2
```

Figure 16: Invoking Operations on a Component Instance

6 Extending Basic Artifacts

Now we have discussed the basics of *Picolo*, let us have a look at how the framework can be extended. As an example, we define a particular version of input port in order to support a non functional requirement of component instances: the trace of invocations. The trace of operations is expected to be used on any input port to trace all operation invocations. Thus, we are going to specialize the `Input` class implementation in order to add the trace mechanism.

6.1 Defining an Extension

Figure 17 presents the `TracedInput` class that implements a trace mechanism as a simple extension to the default behavior of an input port. This implementation is quite straightforward. Most of the behavior is inherited from the default implementation of input ports `picolo.Input`. As we expect the port to print out the information related to an invocation before its actual evaluation, we are redefining the `invoke` method: We first print the information related to the invocation (name of the operation, name of the component instance, and list of

arguments), and then we use the default behavior of the `invoke` method defined in the base class `picolo.Input`.

```
class TracedInput (picolo.Input):
    def __init__ (self, ops=None):
        picolo.Input.__init__ (self, ops)
    def invoke (self, opname, args):
        print '[trace]  ', opname,
        print 'invoked on', self._component.get_name (),
        print 'with args', str (args)
        return picolo.Input.invoke (self, opname, args)
```

Figure 17: Implementation of the Trace Extension for Input Ports

6.2 Setting Up an Extension

The class `TracedInput` is intended to be used in place of the default `Input` one as depicted in figure 18. The unique difference between this example and the previous one (see figure 13) is the use of the `TracedInput` class provided by the `trace` module at instantiation of the input port (instead of the default `Input` class provided by the `picolo` module). Except from that, there is no change in the code of our simple application. The same port substitution is performed on `component2`, in order to trace all the invocations of operations on the input ports of our example.

```
>>> c1 = picolo.Component ('component1')
>>> p1 = trace.TracedInput (ops={'foo': Foo ()})
>>> c1.bind_port('input1', p1)
```

Figure 18: Setting up the Trace Extension

6.3 Using an Extension

Finally, figure 19 presents the result of invoking the operation `foo` on the `component1` input port `input1` that is now traced. Printed output related to the trace module is prefixed by `[traced]`, while printed output related to the application is unprefixed.

```
>>> ref = c1.get_port('input1')
>>> ref.invoke('foo',('raph',))
[trace]  foo invoked on component1 with args ('raph',)
[trace]  hello invoked on component2 with args ('raph',)
hello raph you are caller #1
```

Figure 19: Using the Trace Extension

6.4 Dynamic Handling of Extensions

The use of traced input ports could have been set up without creating a new application: Each input port of the application could have been substituted by a traced input port quite easily. Figure 20 depicts how this substitution can be achieved dynamically. The current port associated to the component `component1` using the name `input1` is unbound, then a new port (that is traceable) is bound to this same component with the same name. As the same set of operation is provided this won't change anything for the users of the component.

```
>>> c1.unbind_port('input1')
>>> p1 = trace.TracedInput (ops={'foo': Foo ()})
>>> c1.bind_port('input1', p1)
```

Figure 20: Dynamic Substitution of Input Port Implementations

In such a port substitution, it is important to control that no output ports are connected to the input port we are updating. If this occurs, it is required to disconnect these ports before the substitution, and to reconnect them afterwards.

7 Describing Components

Using descriptions represents a good means to ease the development of components and component-based applications. Once described, most requirements of a component implementation can be automatically generated. All the technical requirements of a component implementation should be generated, the implementation always following a well defined structure.

Generating the implementation from a well formed description brings several benefits mainly related to the reduction of technical code to be hand written. First, it improves the quality of the implementation as less error are unfortunately introduced. Writing technical code is most of the time error-prone. Second, it reduces the time required to develop a component. Code generation is faster than hand writing.

Regarding descriptions, two levels can be achieved: the description of unitary components and the description of component assemblies. Finally, describing the deployment of component based applications is also an important requirement for a component technology. It represents the basis of automated deployment of applications: A good description and a proper deployment tool is enough to install and launch a complete application.

7.1 Component Type Description

As any software artifact, a component should always be specified, whatever the description means chosen—a UML diagram, a textual or XML definition. In addition to providing a specification, the description of component type defines *component interfaces* that can be used to create several component instances and to automate the use of components.

The free textual syntax we used in figure 9 is an example of component description means. However, its processing is not simple as it requires to write a text parser / compiler. While XML is not a human friendly format, it represents a good description format when dealing with automatic processing. This is partly due to the availability of many XML parsers.

7.1.1 Example of XML Description

Figure 21 presents an example of component specification using XML. The two descriptions specify the properties, the input and output ports of components. They provide both the names and types (operation types are implicitly Python classes defining callable objects) of these elements.

```
<component name="C1">
  <input name="input1">
    <operation name="foo" type="Foo" />
  </input>
  <output name="output2" />
</component>

<component name="C2">
  <input name="input3">
    <operation name="hello" type="Hello" />
  </input>
  <property name="count" type="Counter" />
</component>
```

Figure 21: Descriptions of Component Types

From these two component descriptions, the implementations of the components presented in figures 13 and 12 can be automated. Moreover, a skeleton of the operation classes (as depicted in figures 11 and 10) can be produced to provide you a basis for writing the functional requirements of the component.

7.1.2 Component Descriptions and Component Life-Cycle

The creation of objects relies on the use of a constructor, then you have to write a program in order to instantiate an object. Unlike objects, a component instance life-cycle is ideally described and configured according to its context of use: A vending machine will not serve the same drinks in Göteborg as in Paris. This is due to the expectation for components to be available in 'binary form' and for their use not to rely only on programming but mainly on composition of existing software elements. Finally, the instantiation mechanism is more complex that for objects: various objects representing the component instance have to be created, interconnected and configured.

Once the implementations of operations and properties are available, *Picolo* permits to use such descriptions in two ways: statically, the implementation (like in figure 13) is generated to a file, or dynamically, a factory creates an instance based upon the description. This factory implements the *Factory* design pattern [3] and represents the instantiation mechanism for components. Such a factory

is interesting as it permits the creation of component instances on-demand at runtime (see section 7.3.2).

7.1.3 What About the Others?

Component interface description is more or less available in most technologies: The CCM relies on the use of the OMG Interface Definition Language and to some point to XML grammars: an extension to the Open Software Descriptor format. The web services rely on the use of the Web Service Definition Language [20], and Julia (the Java implementation of Fractal) as well as the EJBs only rely on the use of Java interfaces together with naming patterns.

7.2 Component Assembly Description

Once component types required by an application are specified, it is also important to specify their assembling. The description of component assemblies was first studied in the context of research related to software architectures. An architecture is mainly defined as a set of interconnected components. It provides the global vision of the application.

7.2.1 Example of XML Description

Figure 22 shows an example of assembly description using XML. This description is intended to be used together with the descriptions of component types (figure 21), as it references the component types defined earlier. The assembly description specifies the component instances required by the application together with the connections between them. The assembly states that the application is made of two instances and a connection between them.

```
<assembly>
  <instance name="component1" type="C1" />
  <instance name="component2" type="C2" />
  <connection>
    <source component="component1" port="output2" />
    <target component="component2" port="input3" />
  </connection>
</assembly>
```

Figure 22: Description of a Component Assembly

Like for the evaluation of component interface descriptions (and their prerequisites), a factory can dynamically creates component instances and connect them on the basis of the description, providing a ready to use application. Thus, the availability of operation implementations together with the description of their use and assembling is enough to automate the dynamic creation of application instances.

7.2.2 What About the Others?

Most research results in the field of component assemblies are related to the study of Architecture Definition Languages [11]. Such description means is

unfortunately missing in most industrial (or even general purpose) component technologies. Fractal is the only industrial component model to propose an ADL for describing component architectures. The CCM offers basic XML languages to describe component assemblies.

The others solutions like EJB and web services do not offer such description means. EJBs are mainly running in a centralized application server (even if this server is running on a cluster), and an EJB is mainly a stand alone software element: EJBs are most of the time not connected together. Web services are distributed and intended to be connected together. Work is in progress at the W3C about web services architectures [19] and the composition of web services.

7.3 Component Deployment Description

We have not spoken of non functional properties in the previous descriptors (just functional requirements, component interface descriptions, and structure, component assembly descriptions). Component deployment descriptions may be used to describe how an application should be deployed.

In addition to the instantiation of components and their connections, it may be required to configure the properties and to specify particular non functional requirements to be associated with component instances or connections. These latter are especially useful if you want, for example, to state that the trace mechanism should be activated on particular instances of the application.

7.3.1 Example of XML Description

The component deployment description groups the two previous kind of descriptions in a single XML document[11] as depicted in figure 23. The `lifecycle` module provided with *Picolo* relies on this deployment descriptor to instantiate a component-based application: creating the various component instances, connecting the instances together using their ports, providing access to the main component of the application.

```
<?xml version="1.0" ?>
<picolo name="example1">
  <component name="C1" ... />
  <component name="C2" ... />
  <assembly ... />
</picolo>
```

Figure 23: Description for the Deployment of a Component-Based Application

7.3.2 Deployment of Component-Based Applications

As stated in section 7.1.2, using descriptors permits the automation of component use. On the basis of a descriptor and the implementations of operations and properties, component-based applications can be deployed automatically.

[11] An evolution would be to use several documents referencing each other in order to improve flexibility and reuseness.

This section presents the implementation of the basic factory (provided by the `lifecycle` module) to support deployment.

```
class BasicFactory:
    def create_assembly (self, desc):
        type_defs = dict ()
        for cdesc in desc.get_component ():
            type_defs [cdesc.name] = cdesc
        for inst in desc.get_assembly().get_instance ():
            self.create_component (inst.name,
                                   type_defs [inst.type])
        for cnx in desc.get_assembly().get_connection ():
            src = cnx.get_source ()
            tgt = cnx.get_target ()
            oport = self._instances [src.component] \
                .get_port (src.port)
            iport = self._instances [tgt.component] \
                .get_port (tgt.port)
            cnx = picolo.Connection (oport, iport)
            cnx.connect ()
```

Figure 24: Basic Factory Assembly Creation

Figure 24 presents the implementation of the `create_assembly` method of the basic factory. In a first time, the method build a dict of the various component types (see section 7.1) defined in the deployment descriptor provided as parameter. Then, for each component instance declared in the assembly definition of the descriptor (see section 7.2) the method calls the `create_component` method in order to create a component instance using the declared component type. Finally, for each connection declared in the assembly definition, it retrieves the informations related to the connection (references to the two ports) and establishes it using a `Connection` object.

```
    def create_component (self, name, desc):
        comp = picolo.Component (name)
        for prop in desc.get_property ():
            comp.set_property (prop.name, \
                eval (prop.type +'()', self._locals))
        for elt in desc.get_input ():
            operations = dict ()
            for op in elt.get_operation ():
                operations [op.name] = \
                    eval (op.type + '()', self._locals)
            comp.bind_port (elt.name, \
                picolo.Input (ops=operations))
        for out in desc.get_output ():
            comp.bind_port (out.name, picolo.Output ())
        self._instances [name] = comp
        return comp
```

Figure 25: Basic Factory Component Creation

Figure 25 presents the implementation of the `create_component` method provided by the basic factory. This method creates a component instance on the basis of its description. For each property declared in the component description, an instance is created to represent the value of the property using dynamic evaluation[12] and this instance is added to the component instance list of properties. In a similar way, the input and output ports are created and bound to the component instance. For an input port, all its operations are created and provided to the port at instantiation.

```
>>> import functional
>>> import picorine.core as picorine
>>> import picolo.lifecycle as lifecycle
>>> import picolo.description as description
>>> desc = picorine.factory (description) \
... .create('example1.xml')
>>> factory = lifecycle.BasicFactory (locals ())
>>> factory.create_assembly (desc)
>>> c1 = factory.find_component ('component1')
```

Figure 26: Deploying our Example Application Using the Basic Factory

Figure 26 presents a use of the basic factory provided in module `lifecycle` in order to deploy the application we have defined in this paper. The `functional` module contains the implementation of the operations and properties for our components. The `picorine` module provides a generic loader of XML documents as python objects provided by the `description` module[13]. After loading the description of the application, the factory is created providing the current scope definition (`locals()`), and the assembly of component instances is created by the factory. Finally, the `find_component` method of the factory allows the finding of component instances using their names.

7.3.3 What About the Others?

The CCM offers an XML language to basically describe component deployment, especially how the component instances are located on a network and configured. EJBs also provide deployment descriptor to specify the non functional properties—like persistence and transaction—using XML descriptors. Web services lack this facility, as web services are not considered as a whole but individually.

8 Conclusion

The aim of this paper is not to introduce new ideas about components. It only represents an attempt at providing a basis for introducing component concepts to master students[14] or any software developer interested in such a technology.

[12]The `_locals` attribute contains your local scope when using the factory. This is required to access the modules you have imported.

[13]This module has been generated from a DTD using `picorine` generator.

[14]This work was first designed for master students, however it may fit a bachelor course.

8.1 Contributions and Future Work

The main benefit of the *Picolo* framework is to allow you to dive into the implementation of a component technology, without being overloaded with lots of technical details. Every element of the framework can be analyzed, studied and modified. You can experiment components using *Picolo*, create your own extension, or rewrite your own vision of components providing a complete new implementation. *Picolo* provides a means for a pragmatic introduction to components, and to understand the concepts instead of just understanding a graphical interface provided to *write your first component in 5 minutes*.

This has been achievable thanks to Python for several reasons. First, the framework would not have been so simple using a statically typed language like Java or C++ (a Java implementation of the framework has been partly done). Second, using Python allows you to experiment the use of components in an interactive way: which is an excellent way for seeing what is happening. While being powerful Python eases the development part of studying components: writing an operation is reduced to several lines of code.

Finally, this framework is designed to evolve and to permit you to make it your framework. New extensions can easily be implemented to support the introduction of requirements like distribution or persistence. Anyone can contribute to the *Picolo* framework through extensions or case studies, in order to provide a more complete introduction to component technologies.

8.2 Beyond the Framework

Even if components are bringing lots of benefits to software engineering, all the promises are not reached. The Lego way of plugging things together in order to build new ones is not yet available in the software industry. For example, we are not yet reusing software components as some other fields of the industry reuse component designs like the car industry. We still need to program parts of our applications.

In the meantime, comparing the car industry to the software one is certainly a mistake, as producing a car and producing a piece of software is not the same task. Cars are built through a product line because cars cannot be copied. If thousands of clients want the same, the same car elements will be used to build thousands of car. If thousands of clients need the same piece of software, we can simply copy the binary. If thousands of clients need a specific piece of software, then we have to develop them one by one (even if we can reuse existing pieces).

Finally, research issues are still opened about components. What is the best way to implement and use components? What about a component oriented programming language? Does it make sense? These are some of the questions for which we should find answers, whatever they come from the academia or the industry.

Acknowledgments The author wish to thanks Christophe Dony, Professor at the University of Montpellier (France), with whom the need for such a component model was first discussed at lunch time in late June 2003, and Clémentine Nebut who recently bootstrapped the need for writing this down, and to package and publish the *Picolo* framework.

References

[1] R. Allen, D. Garlan, and J. Ivers. Formal Modeling and Analysis of the HLA Component Integration Standard. In *Proceedings of the Sixth International Symposium on the Foundations of Software Engineering (FSE-6)*, November 1998.

[2] G. Kiczales *et al.* Aspect Oriented Programming. In *Proceedings of 11th European Conference on Object Oriented Programming (ECOOP'97)*, LNCS 1241, pages 220–242, Jyväskylä, Finland, June 1997. Springer Verlag.

[3] E. Gamma, R. Helm, R. Johnson, J. Vlissides, and G. Booch. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Westley Professional Computing, USA, 1995.

[4] G. Heineman and W. Councill, editors. *Component-Based Software Engineering, Putting the Pieces Together.* Addison-Westley, 2001. *ISBN: 0-201-70485-4.*

[5] IS0. *Open Distributed Processing Reference Model – parts 1-4.* International Standard Organization, 1995. ISO 10746-1..4.

[6] C. Lopes and W. Hursch. Separation of concerns. Technical report, College of Computer Science, Northeastern University, Boston, MA, Etats-Unis, February 1995.

[7] R. Marvie. The *Picolo* Framework Home Page, March 2005. `http://www.lifl.fr/~marvie/Software.html#picolo`.

[8] R. Marvie and P. Merle. CORBA Component Model: Discussion and Use with OpenCCM. Technical report, Laboratoire d'Informatique Fondamentale de Lille (LIFL), June 2001.

[9] R. Marvie, P. Merle, and J.-M. Geib. Towards a Dynamic CORBA Component Platform. In *Proceedings of the 2nd International Symposium on Distributed Object Applications (DOA'2000)*, pages 305–314, Antwerpen, Belgium, September 2000. IEEE. *ISBN: 0-7695-0819-7.*

[10] R. Marvie and M.-C. Pellegrini. Modèles de composants, un état de l'art. *Coopération dans les systèmes à objets, Special Issue of l'Objet*, 8(3):61–90, September 2002. *ISBN: 2-7462-0520-3.*

[11] N. Medvidovic and R. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1), January 2000.

[12] ObjectWeb. The fractal project. `http://fractal.objectweb.org/`.

[13] ObjectWeb. Jonas: Java open application server. `http://fractal.objectweb.org/`.

[14] ObjectWeb. Openccm - the open corba component platform. `http://openccm.objectweb.org/`.

[15] D. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communication of the ACM*, 15(12):1053–1058, December 1972.

[16] R. Smith and D. Ungar. Programming as an Experience: The Inspiration for Self. In *ECOOP'95 Conference Proceedings*, Denmark, August 1995.

[17] C. Szyperski and C. Pfister. WCOP'96 Workshop Report. In *Workshop Reader ECOOP'96*, pages 127–130, June 1996.

[18] D. Ungar and R. Smith. Self: The Power of Simplicity. *Lisp and Symbolic Computation*, 4(3):187–205, July 1991.

[19] W3C. *Web Service Architecture, W3C Working Group Note*. World Wide Web Consortium, 2004. `http://www.w3.org/TR/ws-arch/`.

[20] W3C. *Web Service Description Language Version 2.0*. World Wide Web Consortium, 2004. `http://www.w3.org/TR/wsdl20/`.