Separating Concerns with First-Class Namespaces*

Oscar Nierstrasz, Franz Achermann
Software Composition Group, University of Bern
`www.iam.unibe.ch/∼scg`.

Abstract

As applications evolve, it becomes harder and harder to separate independent concerns. Small changes to a software system increasingly affect different parts of the source code. AOP and related approaches offer various ways to separate concerns into concrete software artifacts, but what is the *essence* of this process? We claim that first-class namespaces — which we refer to as *forms* — offer a suitable foundation for separating concerns by offering simple yet expressive mechanisms for defining composable abstractions. We demonstrate how forms help a programmer to separate concerns by means of practical examples in Piccola, an experimental composition language.

1 Introduction

It is well-accepted that complex software systems should be developed as sets of manageable pieces, where each piece ideally addresses a single concern. These pieces are then composed together to achieve the desired behavior of the application. However, carving a complex system up into suitable pieces, may be far from trivial since concerns typically overlap and interfere. Tarr et al. [32] argue that we cannot achieve this separation in a single-paradigm language due to the single dominant dimension of separation supported by the language. For instance, while object-oriented programming separates everything into objects, concerns like persistence and synchronization are not naturally represented as objects, so they get tangled into several objects in the application. The fact that each concern cannot be factored out into a single abstraction leads to components incorporating several varying aspects. This hinders their reusability in other contexts.

AOP and related approaches offer various techniques and mechanisms that make it easier to factor out concerns into composable software artifacts [13]. Many of these approaches attempt to *augment* a host language with features or tools that compensate for the shortcomings of the dominant paradigm.

We propose, instead, to ask a different question, namely:

> Is there a simple programming model that can serve as a good basis for defining arbitrary kinds of composable abstractions?

Such a model should be capable of expressing not only conventional programming abstractions, such as procedures, objects, classes and modules, but also higher-order abstractions, such as (for example) mixins, metaclasses, wrappers, and coordination abstractions.

As a typical example, consider a generic readers/writers synchronization policy. Although it is reasonably straightforward in most programming languages to implement such a policy for a given data abstraction, it can be difficult to impossible to implement a *generic* policy as a composable software artifact that can be applied in a straightforward way to any existing data abstraction.

Furthermore, we seek to raise the level of abstraction so that instead of "wiring" software components together at a low level, we are able to *plug* them together using high-level connectors. A *compositional*

*In *Aspect-Oriented Software Development*, R.E. Filman, T. Elrad, S. Clarke and M. Ak sit (Eds.), pp. 243-259, Addison-Wesley, 2005.

style (also referred to in the literature as an "architectural style" [28]) specifies a set of component *interfaces*, *connectors* to plug together components, and *rules* governing valid compositions of components. In short, we seek to make the way in which software systems are composed explicit and manipulable so they can be easily understood and adapted.

We propose *forms* as a suitable foundation for developing composable software abstractions. Forms unify extensible records, services and first-class namespaces. On one hand they offer a familiar mechanism for modeling various kinds of data abstractions as records. On the other hand they offer a fine degree of control in manipulating, extending and composing the namespaces available to running applications. A *service* is just an abstraction over a form, that is, a function that takes a form as an argument and returns a form. Since a service is also a form, forms offer a suitable basis for modeling high-order software abstractions. Forms can thus be conveniently used to model composable namespaces that both provide and require sets of services.

Piccola [4, 23] is an experimental composition language based on forms. In addition to forms and services, Piccola provides *agents* (concurrent processes) and *channels* (unbounded buffers providing non-blocking `send` and blocking `receive` services) as core mechanisms. With these basic mechanisms, Piccola can be used to easily express the kind of composable abstractions we are targeting.

In Section 2 we provide a brief overview of Piccola and its design rationale. In particular, we show how forms, agents and channels support a layered approach to defining compositional styles. In Section 3 we present a non-trivial example of composing mixin layers with forms. In Section 4 we briefly survey some of the ways in which forms support software composition. We conclude in Section 5 with some remarks concerning the current status of Piccola and ongoing work.

2 Piccola

In this section we give an overview of the layered architecture of Piccola itself, a brief example of generic wrappers in Piccola, and an overview of the kinds of composition abstractions that can be conveniently expressed in Piccola.

Piccola is designed to be a *composition language*, rather than a general purpose programming language. As such, it reduces software composition to a bare minimum of core mechanisms, that is, forms, agents and channels, which can then be used to define higher-level abstractions.

Forms. A *form* is an extensible record. For example, `a=(x=1,y=2)` defines a form `a` that binds labels `x` and `y`. We can *project* a label in a form, such as `w=a.x`, or we can *extend* a form with new bindings; for example, `b=(a,z=3)` extends `a` with a binding for label `z`. A form is also a *namespace*; for example, `('a,x+y)` evaluates `x+y` in the namespace `a`.

Services. A *service* is a function over forms. `println` is a standard Piccola service that prints its argument form. `newPoint p:(x=p.x,y=p.y)` takes a form `p` as its argument and returns a form that extracts just the `x` and `y` bindings from `p`. Note that a service is also a form, so we can bind labels to services, extend services with bindings, or extend forms with services. (A service can be thought of as a form with single "call" label, just as a function object in C++ is an object with an `operator()` member function.)

Agents. The standard Piccola service `run` invokes the `do` service of its argument as a new, concurrent agent. (The code `run(do:println "hello")` creates a new agent that asynchronously prints "hello".)

Channels. A channel is an unbounded buffer that can be used to synchronize agents. `newChannel()` returns a form with services `send` and `receive`. `send` is non-blocking, while `receive` blocks if there is no data on the channel. As a simple example, `stop:newChannel().receive()` is a service that causes an agent to stop dead (*i.e.*, it tries to read from a channel that no other agent ever writes to).

Applications	Components + Scripts
Compositional styles	Streams, events, GUI composition, ...
Standard libraries	Basic coordination abstractions, built-in types
Piccola	Operator syntax, introspection, component wrappers
Piccola-calculus	Forms, agents, channels, services

Table 1: Piccola Layers

The formal semantics of Piccola is compactly expressed with the help of the Piccola-calculus [2, 22], a process calculus that extends Milner's π-calculus [19] with forms and services.

With these mechanisms, Piccola can express three complementary kinds of composition:

1. *Namespace composition.* A form can be *extended* with another form, yielding a new form.
2. *Functional composition.* Services can be *invoked* with a form as an argument, yielding a form as a result.
3. *Agent composition.* Concurrent agents can be composed and *coordinated* by means of shared channels.

2.1 Piccola Layers

Piccola is intended to be used in a layered fashion (see Table 1) to ultimately support a paradigm of "scripting" applications together from a set of software components [6]. In the ideal case, components constitute a kind of "component algebra" in which operators connect components, and again yield components. Scripts, then, compose components, yielding up bigger components.

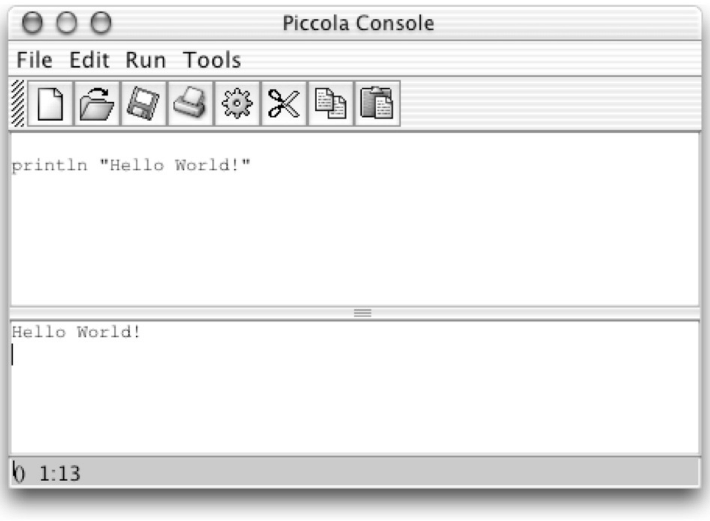
At the lowest level, the Piccola run-time system provides nothing but the core mechanisms of the Piccola-calculus. Programming at this level would be like programming in a concurrent assembly language.

The next level defines the Piccola language. In Piccola, *everything is a form*, so Piccola hides the operators of the Piccola-calculus and models everything in terms of *forms* and *services*. There is no special syntax to express agents and channels, just standard services `run` and `newChannel`. Mechanisms for defining infix and prefix operators are also provided, which is convenient for specifying component connectors as compositional operators. More importantly, reflective mechanisms are provided for exploring forms, wrapping them, and wrapping existing components from a host programming language (*i.e.*, Java or Squeak [11], in the current Piccola implementations).

When Piccola starts up, a number of standard libraries are loaded. At this level, Piccola provides access to a number of built-in types (*i.e.*, Booleans, numbers, strings, collections, file streams, and so on). In most cases, these standard types wrap existing Java components to provide them with more convenient compositional interfaces. The standard libraries also provide a number of standard services that implement various common control structures in terms of forms, agents and channels. Exception handling, for example, is implemented using two agents to run the *try* and *catch* blocks, and a channel to coordinate them in case an exception is raised [6].

On top of the standard libraries, one may define various compositional styles that abstract away from the low-level wiring of the Piccola-calculus, and provide instead higher level plugs, or connectors corresponding to a problem domain. A simple GUI style, for example, that wraps Java AWT and Swing components can easily be defined in Piccola. Furthermore, the style gives us a simple component algebra in which a composition of GUI components is again a GUI component

Finally, at the top level, one may use these styles to script together components. For example, the GUI style is used to build an interactive console for developing and testing JPiccola scripts (see Figure 1).



```
println "Hello World!"
```

```
Hello World!
```

Figure 1: The JPiccola console — scripted from wrapped Java GUI components

2.2 Generic Wrappers

Let us first consider the problem of defining a generic wrapper. `wrapPrePost` wraps each service of its argument form by invoking pre- and post-services before and after the original body. The implementation uses the built-in service `forEachLabel` to iterate over the labels of the argument.

```
wrapPrePost Arg:
  'wrappedForm = newVar()          # local variable
  forEachLabel
    form = Arg.form
    do Label:                      # wrap each service in Arg.form
      wrappedForm.set              # update the result
        'wrappedService Args:
          Arg.pre()                # invoke the pre-service
          Label.project(form)(Args) # invoke original service
          Arg.post()               # invoke the post-service
        wrappedForm.get()          # get the result so far
        Label.bind(wrappedService) # extend it with the new binding
  wrappedForm.get()                # return the wrapped form
```

Although detailed explanation of the code is beyond the scope of this discussion (please see the JPiccola Guide for details [23]), a few observations may help the reader to follow this example. `wrapPrePost` is a service being defined that takes a single `Arg` form as its argument. `Arg` is expected to provide bindings for `pre`, `post` and `form`. `wrappedForm` is a local binding (made local by the Piccola `'` operator). `newVar` is a standard service that returns a persistent variable with `get` and `set` services. `forEachLabel` is another standard service, whose argument is the *indented* form on the following lines. This argument provides bindings for `form` (a value) and `do` (a service). `forEachLabel` generates a first-class representation of each label bound in the form. A first-class label is a form that represents a label and provides services such as `bind`, `project` and `restrict`. The first-class labels are use to reflect over the structure of the argument form and build a new, wrapped representation.

A key point to notice is that all services in Piccola are *monadic*; that is, they always take a single form as an argument, rather than a tuple of forms. This makes the task of wrapping services much simpler than it would be in most programming languages.

We could now use this generic wrapper to wrap a component with an arbitrary synchronization policy. Suppose, for example, we define semaphores like this:

```
newSemaphore:
  'sem = newChannel()
  p: sem.receive()
  v: sem.send()
  'v()
```

and a mutual exclusion synchronization policy like this:

```
newMutexPolicy:
  'sync = newSemaphore()
  pre:  sync.p()
  post: sync.v()
```

Now if `F` is some form, we can wrap each of its services with a mutual exclusion policy as follows:

```
MutexF = wrapPrePost
  form = F
  newMutexPolicy()
```

A readers/writer synchronization policy could be similarly defined. In this case, however, we must distinguish between reader and writer services, and use the `wrapPrePost` service to wrap them separately with their own policies:

```
bindRWPolicy Arg:
  wrapPrePost
    form = Arg.reader
    pre  = Arg.policy.preR
    post = Arg.policy.postR
  wrapPrePost
    form = Arg.writer
    pre  = Arg.policy.preW
    post = Arg.policy.postW
```

We must now explicitly list the services to be wrapped:

```
RWsynchedF = bindRWPolicy
  policy = newRWPolicy()           # create reader writer policy
  reader = (r1 = F.r1, r2 = F.r2)  # r1 and r2 are reader methods
  writer = (w = F.w)               # w is a writer method
```

Generic wrappers play an important part in compositional styles, since they constitute a form of reusable "glue code" that can adapt components to different styles. Automatically invoked wrappers are used in Piccola, for example, to adapt Java components to compositional styles. Java AWT and Swing components, for example, are automatically wrapped when they are accessed by a Piccola script, allowing them to be connected using the GUI style defined in Piccola.

3 An Example: Mixin Layer Composition

In this section, we give a concrete example of mixin layer composition [30] implemented as a compositional style in SPiccola (the Squeak implementation of Piccola [25]). Mixin layers are (in our view) a less well known and non-trivial composition style. Implementing mixin layers requires an object-oriented language that supports nested classes and mixins. The language P++, for example, extends C++ to support static and type-safe mixin layer composition [29]. Implementing mixin layer composition in Piccola thus serves to validate that Piccola is expressive enough to tackle high-level composition abstractions. Finally, mixin layers are a good candidate to illustrate component algebras, because composed mixin layers are again mixin layers.

We present the graph traversal application proposed by Holland [10]. This application defines different operations on an undirected graph. *VertexNumbering* numbers the nodes in a depth-first order, *CycleChecking* determines whether the graph contains a cycle, and *ConnectedRegions* partitions the nodes of the graph into connected regions. Holland implemented the application based on a framework. Later, Van Hilst et al. [33] reimplemented it using roles and mixins. Smaragdakis and Batory finally used mixin layers to implement the same application [30, 31].

The three main implementation classes are `Graph`, `Vertex`, and `Workspace`. The `Graph` class defines a container of vertices with the usual graph properties. The nodes are stored as instances of the class `Vertex`. The `Workspace` class includes the specific part of a traversal. For instance, the `Workspace` object plays the role *WorkspaceNumber* in the *VertexNumbering* application to associate numbers to the nodes. This role specifies a slot in which to store a current number and to assign and increment this number each time a new node is visited during depth-first traversal. We can implement such a role using a mixin. The mixin adds the specific members and operations to its superclass when composed. Similarly, a mixin adds the number slot to a vertex class.

Smaragdakis and Batory use the GenVoca model [9] to keep the different mixins applied to classes in synch. A GenVoca component is a mixin layer. In essence, a mixin layer encapsulates all the mixins necessary for a single collaboration. For instance the mixin layer *Number* to implement the *VertexNumbering* collaboration contains two mixins: one to add the vertex numbering during traversal and one to add the number to a vertex. The advantages of using mixin layers instead of isolated mixins are clear: Design or change elements in the application are encapsulated and implemented in a single component.

3.1 Mixins Layers in Piccola

We now seek a simple way to model and compose mixins and mixin layers. We would like to achieve the kind of simplicity illustrated by the following example. `graph` provides constructors for graphs and vertices. We then compose it, using the `**` mixin layer composition operator, with mixin layers `dft`, `numberNodes` and `cycle`, which, respectively, provide services for depth-first traversal, automatic numbering of nodes, and cycle detection:

```
layers = graph ** dft ** numberNodes ** cycle
newGraph: layers.asGraph()
newVertex: layers.asVertex()
g = newGraph()
...
```

We claim that the model of explicit namespaces offered by forms provides us with a good way of modeling compositional abstractions like mixin layers.

First of all, although Piccola provides only forms, not objects or classes, it is relatively simple to model objects and classes as forms. An object is just a form providing services that access some private state, such as the semaphores we saw earlier. A class is a form offering services shared by all instances, such as constructors, and services to create subclasses [6].

In our example, `graph` represents a base-level mixin layer, that is, one that provides services `asGraph` and `asVertex` to create new graphs and vertices. A graph itself provides services like `insert` and `each` to insert a new vertex or visit each vertex. The other mixin layers wrap the `asGraph` and `asVertex` of the layer below to create graphs and vertices with new or adapted services. In each case, it is important that the mixin layer *simultaneously* wrap both constructors, since the new and adapted services typically depend on each other.

How do the mixin layers work? Let consider the `numberNodes` layer

```
numberNodes =
  asVertex V:
    V
    number = newVar(0)
  asGraph G:
    G
    visit:
      n = newCounter()
      G.each(do V: V.number.set(n.inc()))
```

`asVertex` wraps a vertex to provide it with a number. `asGraph` wraps a graph to provide it with a `visit` service that increments the `number` of each of its vertices. These two services simultaneously wrap vertices and graphs from the layer below, so we are sure that when a vertex is visit, it actually provides a `number` binding.

Next, we need to implement a composition operator `**` such that "`A ** B`" is a composite mixin layer, provided that `A` and `B` are mixin layers. We could implement a generic mixin layer composition operator in the style of our `wrapPrePost` service, but for simplicity we just consider the specific problem of composing graph mixin layers:

```
'Defaults = (asVertex X:X, asGraph X: X)
_**_ A B:
  'A = (Defaults, A) # possibly override defaults
  'B = (Defaults, B)
  asVertex X: B.asVertex(A.asVertex(X))
  asGraph X: B.asGraph(A.asGraph(X))
```

The form `Defaults` contains default values for the `asVertex` and `asGraph` wrappers, namely the identity function. We rebind `A` and `B` so that the actual arguments may override these defaults. Finally, we compose new `asVertex` and `asGraph` wrappers from those provided by the arguments.

Note that the order in which the mixin layers are composed affects the end result. `numberNodes`, for example, depends on the `each` service provided by graphs of the layer below. The `graph` layer provides such a service, but the `dft` layer happens to replace this service by a depth-first traversal `each` service. As a consequence, `graph**dft**numberNodes` and `graph**numberNodes**dft` exhibit different behaviors.

3.2 Software Evolution with Mixin Layers

The resulting separation of concerns enables us now to combine or replace mixin layers in a straightforward way. Suppose, for example, that graphs may be exposed to concurrent clients. In this case, we might want to apply a synchronization policy to the graph. The mixin layer `exclusive` applies a mutual exclusion synchronization policy to all the methods of graph. Since the order of mixin layer composition is significant, we apply this layer last of all:

```
layers = graph ** dft ** numberNodes ** cycle ** exclusive
```

We can also change the depth-first traversal to a breadth-first traversal by replacing a component:

```
layers = graph ** bft ** numberNodes ** cycle
```

We can adapt a layer that (by chance) does not follow the naming conventions by introducing some *glue code*:

```
myLayer =
  asGraph = legacyLayer.addFancyFeatureToGraph
  asVertex = legacyLayer.addFancyFeatureToNode
```

In case there are many components that need to be adapted in the same way, we can abstract from the glue code to obtain a general-purpose *glue abstraction*:

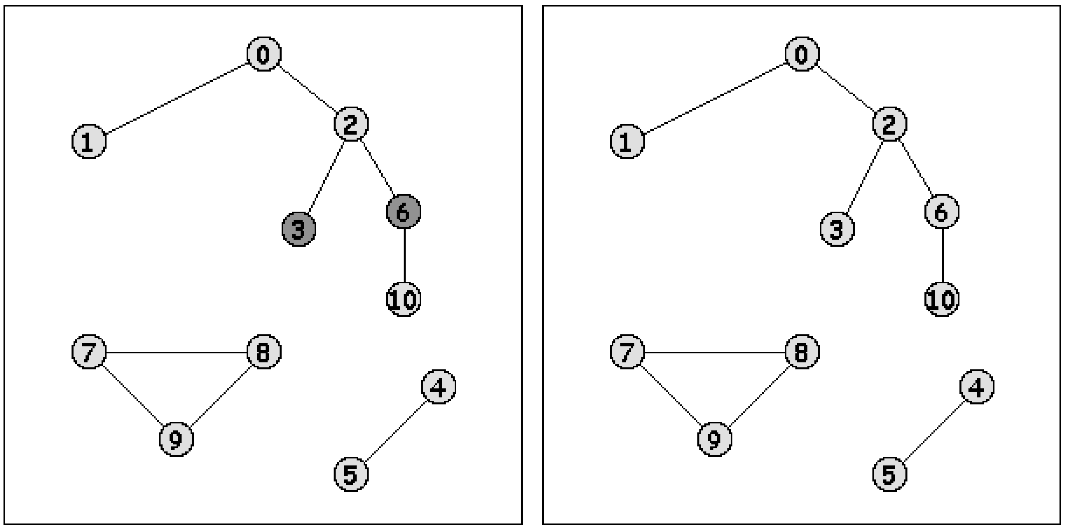


Figure 2: `graph() ** dft ** visual` vs. `graph() ** sdft ** visual`

```
legacyAdaptor legacyLayer:
  asGraph = legacyLayer.addFancyFeatureToGraph
  asVertex = legacyLayer.addFancyFeatureToNode
myLayer = legacyAdaptor(myLegacyLayer)
```

As a final validation of mixin layers, we found it straightforward to extend the graph traversal mixin layer framework with a new mixin layer for visualizing graphs. The `visual` layer in SPiccola packages the glue code needed to display graphs in Squeak:

```
visual =
  asGraph G:
    G
    'defaultColor = Smalltalk("Color").yellow()
    morph = newVar(0)
      ...
    # replace G.each by an animated version
    each Block:
      ...
```

This new mixin layer can now be added to any graph traversal mixin layer composition at the appropriate point. In figure 2 we see visualizations of thread-unsafe and thread-safe graphs that have been subjected to two concurrent traversals that mark and unmark the nodes (*i.e.* by changing their color). The visualization clearly shows that only the thread-safe graph is uniformly colored at the end.

4 Specifying Compositional Styles with Forms

We now briefly survey some of the typical compositional styles and techniques that are used to make software more flexible and adaptable. We demonstrate how Piccola forms support them. Note that these styles are not orthogonal.

Component algebras. A *component algebra* is a compositional style in which the composition of two or more components is again a component. The best-known example of a component algebra is pipes and filters. The components are sources, filters and sinks, and the principle operator is the pipe. A source composed with a filter yields a source, and a filter composed with a filter is again a filter.

We believe that any compositional style can be conveniently expressed as a component algebra. In addition to pipes and filters, we have developed and experimented with component algebras for GUI components [6], input/output streams [4], coordination of concurrent agents [3], and mixin layers [21].

Higher-order wrappers. Many kinds of extensions can be factored out as simple wrappers, adding "before and after" behavior. Since all values (including abstractions) are forms in Piccola, abstractions are higher-order, making it easy to specify higher-order wrappers. As we have seen in section 2.2, services in Piccola are *monadic*, always taking a single form as an argument. This makes it possible to define *generic wrappers* that do not depend on the number of arguments. Piccola uses wrappers heavily to adapt components to conform to a particular style.

Glue abstractions. Many glue abstractions can be expressed as simple wrappers. Glue abstractions can wrap known services or add new ones while leaving other undisturbed [16, 26]. A simple glue abstraction, for example, can wrap a java.math.BigInteger to provide it with the usual arithmetic operators. (Java does not provide operator overloading, so a Biginteger provides methods like add and multiply instead of + and −.)

Mixins and metaobjects. Higher-order wrappers make it possible to define mixins and other composition mechanisms for building objects. Piccola provides only forms as "primitive objects", but one can define a variety of other object models on top of forms [26]. One of Piccola's few keywords is `def`, used to define a fixpoint, but it is also possible to delay binding of self, which makes object models with explicit metaobjects attractive. Metaobjects enable run-time reflection [12].

Coordination abstractions. Piccola provides primitives to instantiate concurrent agents or to explicitly create new channels within scripts. The formal semantics of Piccola is in terms of a process calculus, so concurrency is built-in, not added-on [16, 26]. This makes it easy to define coordination abstractions as abstractions over scripts. Furthermore, coordination can be seen as a special case of scripting, and many coordination styles can be naturally expressed as component algebras [3].

Implicit policies. Forms are also used in Piccola to represent *namespaces* [5]. Whenever a script is evaluated, it has access to two special namespaces, representing respectively the *root* context and the *dynamic* context. The root context defines the global environment, but can be specialized to define a "sandbox" for an untrusted agent, or to override or extend global services (like `println`). The dynamic context is the environment provided by a client of an agent, and can be used to define implicit policies. This mechanism can be used, for example, to define an exception handling mechanism for Piccola [5, 6] (the handler is always passed in the dynamic context). The same mechanisms are used more generally to optionally override any kind of default policy.

Default arguments. Since abstractions are monadic, taking a single form as an argument, and since forms can be extended, it is straightforward to define default arguments for services, as we did in defining the `**` mixin layer composition operator.

5 Concluding Remarks

As we have argued elsewhere [4], Piccola lies somewhere between a *scripting language*, like Python or TCL, an *architectural description language* (ADL), like Wright [7] or Rapide [15], a *coordination language*, like Darwin [18] or Manifold [8], and a *glue language*, like Smalltalk or C.

A type system has been developed for Piccola [16], but it too is at the level of the process calculus. We would like to reason about higher-level types in terms of components and their composition. Ideally, we might like a type system that can express not only required and provided services, but even some more detailed dependencies [14, 20, 24].

In previous papers, we have presented the conceptual framework of *components*, *scripts* and *glue* [27], the formal underpinnings of Piccola in terms of the Piccola-calculus [4, 17, 23], and a tour of the Piccola language features [6]. We have demonstrated how Piccola forms can model different notions of *explicit namespaces* [5], we have shown how different forms of *coordination* can be expressed as compositional styles [3], and we have argued that aspect-oriented programming [13] can be expressed as *feature mixins* in Piccola [1]. We have also argued that software systems can evolve gracefully only if they are designed in such a way as to cleanly separate stable and flexible aspects into *components* and *scripts* [21].

Here we have argued that the mechanism of explicit namespaces provided by forms is crucial to achieving a clean separation of concerns.

Piccola is designed to be a *composition language*, good at expressing different kinds of *compositional styles*, each of which may be suitable for composing components for different application domains. We are still experimenting with applications of Piccola. Although we believe that Piccola provides the right abstractions needed to express applications as flexible compositions of software components, we still have to prove that these techniques can succeed in separating concerns for complex domains where other approaches have failed.

Our long-term goal is to develop a framework that supports the definition of higher-level composition operators and in which we can reason about properties of composite components. Piccola should serve as a platform to develop a composition environment hosting components and supporting the flexible scripting of components within user-defined architectural styles.

Acknowledgements

This work has been supported by the Swiss National Science Foundation under Projects #20-53711.98, "A framework approach to composing heterogeneous applications", #20-61655.00, "Meta-models and Tools for Evolution Towards Component Systems", and the Swiss Federal Office for Education and Science under Project BBW #96.00335-1, within the Esprit Working Group 24512: "COORDINA: Coordination Models and Languages."

Thanks to Robert Filman and Markus Gaelli for numerous suggestions and corrections.

References

[1] Franz Achermann. Language support for feature mixing. In *Workshop on Multi-Dimensional Separation of Concerns in Software Engineering (ICSE 2000)*, Limerick, Ireland, June 2000.

[2] Franz Achermann. *Forms, Agents and Channels — Defining Composition Abstraction with Style.* PhD thesis, University of Berne, January 2002.

[3] Franz Achermann, Stefan Kneubühl, and Oscar Nierstrasz. Scripting coordination styles. In António Porto and Gruia-Catalin Roman, editors, *Coordination '2000*, volume 1906 of *LNCS*, pages 19–35, Limassol, Cyprus, September 2000. Springer-Verlag.

[4] Franz Achermann, Markus Lumpe, Jean-Guy Schneider, and Oscar Nierstrasz. Piccola — a small composition language. In Howard Bowman and John Derrick, editors, *Formal Methods for Distributed Processing — A Survey of Object-Oriented Approaches*, pages 403–426. Cambridge University Press, 2001.

[5] Franz Achermann and Oscar Nierstrasz. Explicit Namespaces. In Jürg Gutknecht and Wolfgang Weck, editors, *Modular Programming Languages*, volume 1897 of *LNCS*, pages 77–89, Zürich, Switzerland, September 2000. Springer-Verlag.

[6] Franz Achermann and Oscar Nierstrasz. Applications = Components + Scripts — A Tour of Piccola. In Mehmet Aksit, editor, *Software Architectures and Component Technology*, pages 261–292. Kluwer, 2001.

[7] Robert Allen and David Garlan. The Wright architectural specification language. CMU-CS-96-TB, School of Computer Science, Carnegie Mellon University, Pittsburgh, September 1996.

[8] Farhad Arbab. The IWIM model for coordination of concurrent activities. In Paolo Ciancarini and Chris Hankin, editors, *Proceedings of COORDINATION '96*, volume 1061 of *LNCS*, pages 34–55, Cesena, Italy, 1996. Springer-Verlag.

[9] Don Batory, Vivek Singhal, Jeff Thomas, Sankar Dasari, Bart Geraci, and Marty Sirkin. The GenVoca model of software-system generators. *IEEE Software*, pages 89–94, September 1994.

[10] Ian M. Holland. Specifying reusable components using contracts. In O. Lehrmann Madsen, editor, *Proceedings ECOOP '92*, volume 615 of *LNCS*, pages 287–308, Utrecht, the Netherlands, June 1992. Springer-Verlag.

[11] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of Squeak, A practical Smalltalk written in itself. In *Proceedings OOPSLA '97*, pages 318–326. ACM Press, November 1997.

[12] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[13] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings ECOOP '97*, volume 1241 of *LNCS*, pages 220–242, Jyvaskyla, Finland, June 1997. Springer-Verlag.

[14] Stefan Kneubühl. Typeful compositional styles. Diploma thesis, University of Bern, April 2003.

[15] David C. Luckham and James Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, September 1995.

[16] Markus Lumpe. *A Pi-Calculus Based Approach to Software Composition*. Ph.D. thesis, University of Bern, Institute of Computer Science and Applied Mathematics, January 1999.

[17] Markus Lumpe, Franz Achermann, and Oscar Nierstrasz. A Formal Language for Composition. In Gary Leavens and Murali Sitaraman, editors, *Foundations of Component Based Systems*, pages 69–90. Cambridge University Press, 2000.

[18] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeffrey Kramer. Specifying distributed software architectures. In *Proceedings ESEC '95*, volume 989 of *LNCS*, pages 137–153. Springer-Verlag, September 1995.

[19] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part I/II. *Information and Computation*, 100:1–77, 1992.

[20] Oscar Nierstrasz. Contractual types. Technical Report IAM-03-004, Institut für Informatik, Universität Bern, Switzerland, 2003.

[21] Oscar Nierstrasz and Franz Achermann. Supporting Compositional Styles for Software Evolution. In *Proceedings International Symposium on Principles of Software Evolution (ISPSE 2000)*, pages 11–19, Kanazawa, Japan, November 2000. IEEE.

[22] Oscar Nierstrasz and Franz Achermann. A calculus for modeling software components. In S. Graf F. S. De Boer, M. M. Bonsangue and W-P. de Roever, editors, *FMCO 2002 Proceedings*, volume 2852 of *LNCS*, pages 339–360. Springer-Verlag, 2003.

[23] Oscar Nierstrasz, Franz Achermann, and Stefan Kneubühl. A guide to JPiccola. Technical Report IAM-03-003, Institut für Informatik, Universität Bern, Switzerland, June 2003.

[24] Oscar Nierstrasz, Jean-Guy Schneider, and Franz Achermann. Agents everywhere, all the time. In *ECOOP 2000 Workshop on Component-Oriented Programming*, 2000.

[25] Nathanael Schärli. Supporting pure composition by inter-language bridging on the meta-level. Diploma thesis, University of Bern, September 2001.

[26] Jean-Guy Schneider. *Components, Scripts, and Glue: A conceptual framework for software composition.* Ph.D. thesis, University of Bern, Institute of Computer Science and Applied Mathematics, October 1999.

[27] Jean-Guy Schneider and Oscar Nierstrasz. Components, scripts and glue. In Leonor Barroca, Jon Hall, and Patrick Hall, editors, *Software Architectures — Advances and Applications*, pages 13–25. Springer-Verlag, 1999.

[28] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice-Hall, 1996.

[29] Vivek P. Singhal. *A Programming Language for Writing Domain-Specific Software System Generators.* PhD thesis, University of Texas at Austin, September 1996.

[30] Yannis Smaragdakis and Don Batory. Implementing layered design with mixin layers. In Eric Jul, editor, *Proceedings ECOOP '98*, volume 1445 of *LNCS*, pages 550–570, Brussels, Belgium, July 1998.

[31] Yannis Smaragdakis and Don Batory. Implementing reusable object-oriented components. In *5th International Conference on Software Reuse*, Victoria, Canada, June 1998.

[32] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N Degrees of Separation: Multi-dimensional Separation of Concerns. In *Proceedings of ICSE '99*, pages 107–119, Los Angeles CA, USA, 1999.

[33] Michael VanHilst and David Notkin. Using C++ Templates to Implement Role-Based Designs. In *JSSST International Symposium on Object Technologies for Advanced Software*, pages 22–37. Springer Verlag, 1996.