Separation of Concerns through Unification of Concepts[1]

Oscar Nierstrasz, Franz Achermann
Software Composition Group, University of Berne[2]

Abstract. Separation of concerns is a principle we apply to reduce complexity. This principle is especially important when it is used to separate *stable* from *flexible* parts of software systems to reduce the complexity of software evolution. We encapsulate the stable parts as *components* and the flexible parts as *scripts*. But there is a large range of requirements and consequent techniques available to achieve this separation. We propose a simple, unifying framework of *forms*, *agents*, and *channels* for modelling components and scripts. We have also developed an experimental *composition language*, called Piccola, based on this framework, that supports the specification of applications as flexible compositions of stable components.

1. Introduction

As software engineers, we separate concerns in order to reduce the complexity of designing, constructing and maintaining large software systems. Some complexity is inherent in the requirements of large software systems, but, we argue, *the worst forms of complexity arise from software evolution*. Tangled code poses a problem only if that code needs to be modified or extended. This, however, is the case for any interesting piece of software! As a consequence, we suggest that the biggest gains are to be achieved by concentrating on *reducing the complexity of software change*.

In fact, we see that the tools and techniques that achieve the biggest gains in productivity address precisely this issue: very high-level languages, software components, and fourth generation development environments raise the level of abstraction by *separating what is stable from what is not*, and thereby make it easier to introduce changes. We propose, therefore:

Applications = Components + Scripts

as a guiding principle for separation of concerns. *Components* wrap up provided (and required) services behind a standard interface, and generally represent the *stable* parts of applications. *Scripts* plug components together, and represent the *flexible* parts of applications. Sometimes components embody a mix of stable and unstable aspects, in which case components themselves can be scripted.

---

1. Presented at the ECOOP 2000 Workshop on Aspects & Dimensions of Concerns.

2. Institut für Informatik (IAM), Universität Bern, Neubrückstrasse 10, CH-3012 Berne, Switzerland. {oscar,acherman}@iam.unibe.ch. www.iam.unibe.ch/~scg.

The scriptable components paradigm additionally assumes that *components have been designed to be plugged together*, according to a particular "compositional style."[1] Furthermore, scripts may make use of special connectors to plug components together, namely *coordination abstractions* to mediate the connections between components, and *glue abstractions* to mediate between components whose plugs do not match [11].

To support this paradigm, we have developed Piccola, a small language for specifying components, connectors and scripts [1][3]. The goal of Piccola is to facilitate the specification of components, connectors and scripts corresponding to different compositional styles. As such, the kinds of abstraction mechanisms required by Piccola are somewhat different than those needed for a general-purpose language. In order to have the simplest possible framework to define compositional styles, we choose a small set of primitives that unify various concepts:

- *Forms* embody *structure*. Forms are immutable records that can be polymorphically extended with additional bindings (yielding a new form). Forms unify *objects*, *keyword-based arguments* and *namespaces*.
- *Agents* embody *behaviour*. Agents are concurrent, communicating entities whose behaviour is specified by a script. Agents implement the connections between components. Agents unify *concurrency* and *composition*.
- *Channels* embody *state*. Channels are the mailboxes that agents use to communicate. They unify *synchronization* and *communication*.

In section 2 we briefly summarize our requirements for software evolution from an open systems perspective. In section 3 we will survey a series of programming idioms and techniques that exploit separation of concerns to promote software flexibility. We conclude with some remarks concerning ongoing research. For the sake of brevity, we include Piccola examples, but instead refer to several other publications containing extensive examples.

2. Open Systems Requirements

Successful software systems are doomed to either change [7] or quickly become obsolete. We can classify some of the more typical kinds of change:

- *Change in business logic.* New functionality may be added, existing functionality modified or reconfigured, or new policies may be put into place.
- *Change in platform.* The operating system, database, user interface, standard libraries or even programming language may change.
- *Change in clients.* The system may need to offer its services to new, or different kinds of clients, or may need to be "integrated" with other new or existing systems.

In each case, "tangled" (i.e., strongly coupled) or "scattered" (i.e., weakly cohesive) code can lead to a maintenance nightmare. The solution, very broadly, is to factor out the tangled aspects in such a way that they can be easily recomposed in a more flexible way. "Factoring out"

1. I.e., an "architectural" style in terms of components, connectors and composition rules [12]. Since we are not always scripting at the level of architectures, we prefer the more modest term, "compositional style".

aspects can be done in a large number of different ways, however. Let us briefly consider a few typical examples:

- *Multi-language support.* Successful software designed for a specific target language may have to be adapted to support new languages. The user dialogues are tangled with all of the user interface code. Two typical approaches are to either (i) *generate* the UI code, parameterized by the target language, or (ii) *delegate* all dialogues to language-specific dialogue manager. The former technique is static, whereas the latter supports dynamically swapping languages.
- *GUI look and feel.* Multi-platform applications are often required to support the look and feel of the host platform. The look and feel can be considered a *policy* which must be separated from the GUI mechanisms delivered by an abstract interface. Depending on the implementation technique, the policy may be statically or dynamically specified.
- *Compiler extensions.* Extensions to a programming language or to the tools that support the language often entail changes and extensions to the way in which syntactic entities are represented. These changes may *cross cut* different parts of the compiler (i.e., those that parse and generate symbolic representations, and those that process them). In such cases, inheritance and generics are of limited help because they are purely static techniques. There are typically no easy solutions to such problems, but *wrappers* and *mixins* often help in situations where new functionality should be added uniformly to existing objects and classes.
- *Concurrency.* A functioning sequential system cannot always be easily converted to work in a concurrent context. The safest solution is to wrap the whole system up as a monitor, but sometimes more fine-grained concurrency control is required. Most programming languages, unfortunately are not expressive enough to allow synchronization or coordination abstractions to be separately specified from the synchronized or coordinated entities. In Java, for example, synchronization code is always tangled with computational code. In order to wrap an existing Java class with, for example, a readers/writers mutual exclusion policy, one is forced to write a lot of boilerplate code. *Higher-order wrappers*, some form of *reflection* (introspection, as provided by Java's "reflection" package, is typically not enough), or a *meta-object protocol* are techniques that help to factor out such aspects.
- *Heterogeneous systems policies.* There is a long list of aspects that arise in heterogeneous and distributed software systems that must be factored out as policies: persistence, transactions and security are three common examples [6].

From this short list of examples, we already identify a few requirements:

- *Open object model.* A simple, but open object model will better support the specification of needed abstractions than a rich, but fixed object model
- *Run-time reflection.* Introspection is not enough.
- *Run-time composition.* Static composition and extension mechanisms are not enough.

3. Language Support for Separation of Concerns

There are various idioms and techniques that apply the principle of separation of concerns in order to make software flexible and adaptable. Let us examine several of these in turn, and see how Piccola's unifying concepts of forms, agents and channels support these techniques.

Component algebras. We know how to specify components. We have more difficulties specifying how to compose them. Instead of using low-level *wiring*, we want to *plug* components together. We propose that the right way to think about plugging components is in terms of a *component algebra*. The sorts of such an algebra are different kinds of components, each characterized by different plugs and sockets that represent required and provided services. The operators of the algebra are the connectors. A script, then, is just an expression that composes components, where each subexpression is also a component [1]. Unix pipes and filters are the standard example of this way of expressing a compositional style. We claim that most styles can benefit from this approach [1][3][4][11].

In Piccola, component services are encapsulated as forms. Different sorts of components are represented as forms with different sets of labels. Connectors are user defined operators realized either as abstractions available in the local context, or as special services of a component. (A form is a kind of "primitive object," and its services are its methods.) A script composes forms, and yields a new form.

Higher-order wrappers. Many aspects can be factored out as simple wrappers, adding "before and after" behaviour. In Piccola, all values are forms, including abstractions. (An abstraction 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.) Abstractions are therefore higher-order, making it easy to specify higher-order wrappers. Furthermore, abstractions are *monadic*, always taking a single form as an argument. This makes it easy to define *generic wrappers* that do not depend on the number of arguments.

Glue abstractions. Many glue abstractions can be expressed as simple wrappers. Forms in Piccola can be *polymorphically extended* (i.e., by record concatenation), so glue abstractions can wrap known services or add new ones while leaving other undisturbed [8][10].

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 [10]. One of Piccola's few keywords is `def`, used to define a fixpoint, but it is also possible to delay binding of self [10], which makes object models with explicit metaobjects very attractive. Metaobjects enable run-time reflection [5].

Coordination abstractions. Piccola provides lightweight 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 [8][10]. 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 [4].

Implicit policies. Forms are also used in Piccola to represent *namespaces* [2]. 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 print). 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 [1][2] (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 forms can be polymorphically extended, it is easy to define default arguments for services. (An agent just appends the received argument to the defaults.) This makes it easy to extend services with new options without breaking existing clients.

4. Research

Piccola has borrowed ideas from many different sources. Its innovation, we feel, lies mainly in the way it brings a few, simple concepts together that, in combination, support the paradigm, "Applications = Components + Scripts."

Piccola is still an experimental language. There currently exist implementations on three different platforms, each with its own syntax. A preferred syntax is emerging, but there are still open questions concerning usability and expressiveness.

A formal semantics in terms of process calculus exists, but this is too low-level to support reasoning. We would like to reason about applications at the level of their composition. That is, if we know that components conforming to a particular compositional style have certain properties, then valid compositions will also have certain interesting properties.

A type system has been developed for Piccola [8], 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 [9].

Finally, 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.

5. References

[1] Franz Achermann and Oscar Nierstrasz, "Applications = Components + Scripts — A tour of Piccola," *Software Architectures and Component Technology*, Mehmet Aksit (Ed.), Kluwer, 2000, to appear.

[2] Franz Achermann and Oscar Nierstrasz, "Explicit Namespaces," to appear, JMLC 2000.

[3] Franz Achermann, Markus Lumpe, Jean-Guy Schneider and Oscar Nierstrasz, “Piccola - a Small Composition Language,” *Formal Methods for Distributed Processing, an Object Oriented Approach*, Howard Bowman and John Derrick. (Eds.), Cambridge University Press., 2000, to appear.

[4] Franz Achermann, Stefan Kneubuehl and Oscar Nierstrasz, “Scripting Coordination Styles,” April 2000, submitted for publication.

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

[6] Doug Lea, “Design for Open Systems in Java,” *Proceedings COORDINATION’97*, David Garlan & Daniel Le Mètayer (Ed.), Springer-Verlag, Berlin, Germany, September 1997, pp. 32-45.

[7] Lehman M. M. and Belady L., *Program Evolution - Processes of Software Change*, London Academic Press, 1985.

[8] 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.

[9] Oscar Nierstrasz, Jean-Guy Schneider and Franz Achermann, “Agents Everywhere, All the Time,” 2000, Submitted to the joint ECOOP 2000 Workshops on Component-Oriented Programming and Pervasive Component Systems.

[10] 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.

[11] Jean-Guy Schneider and Oscar Nierstrasz, “Components, Scripts and Glue,” *Software Architectures — Advances and Applications*, Leonor Barroca, Jon Hall and Patrick Hall (Eds.), pp. 13-25, Springer, 1999.

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