Composition Languages for Black-Box Components

Position Paper

Roel Wuyts
Software Composition Group
Institut für Informatik
Universität Bern, Switzerland
roel.wuyts@iam.unibe.ch

Stéphane Ducasse
Software Composition Group
Institut für Informatik
Universität Bern, Switzerland
ducasse@iam.unibe.ch

ABSTRACT

Supporting reuse of existing pieces of code is one of the main goals of software engineering. In the name of reuse, module-based programming languages came to be, only to be surpassed by object-oriented technology. With the same motivation component-based solutions are overtaking object-oriented solutions. However, the delegation-only focus of component-based programming risks of resulting in the same problems that modular-based approaches ran into. To counter this, we claim that one of the important problems that should be addressed by component languages is the *composition* of components. More specifically, we see component languages where components are black-box abstractions, and with (one or more) composition languages to glue them together. As an example we show a functional (Piccola) and a logic (QSoul) composition approach.

1. INTRODUCTION

Years ago, when object-oriented programming was still partially uncharted territory, several questions were floating around to be answered: what is an object ? what is a class ? How can we compose objects ? How can we compose classes ? What runtime support is needed ? What's the semantic of polymorphism ? What's the semantic of a *super call* ? Several languages existed that implemented solutions for these -and other- important questions. Some of these were purely object-oriented (think Simula or Smalltalk), some of them were additions to procedural languages (think C++). Some were completely new kinds (think Self), or offered composition mechanisms such as mixins and mixin based approaches [3, 13]. The important aspect is that each of these languages made (implicitly or explicitly) certain choices, and allowed to test these choices on real code and real problems. For example, multiple inheritance as introduced by C++ proved to have too much conceptual and was not introduced in other mainstream object-oriented languages.

Now, in the beginning of the next millennium, analogous questions pop up in relation to components: what is a component ? How to compose components ? What are the exact semantics for certain composition mechanisms ? These questions are imported to answer, and will eventually give rise to a new breed of programming languages: component languages.

In this paper we have a look at the move from module-based programming languages to object-orientation. Then we have a quick glance at current component systems and the composition mechanisms they support. Finally we look at 'pure' component languages, and the issues that play a role in their definition, using the lessons learned during from the move of module-based languages to object-oriented languages into account. More specifically we claim that one of the issues that is crucial for component languages is the *composition mechanism* that is offered to compose (or script or glue) components. We give two examples of composition languages that could be used in component languages.

2. FROM MODULES TO OBJECTS

Before the advent of the component-oriented paradigm, most languages had a notion of a *module*. The goal of a module was one of packaging: making subsystems independent of other subsystems. It became thus easier to exchange subsystems for one another, whenever their interfaces were compatible, and to develop and test parts of the system independent of other parts of the system. Most procedural-based languages (such as ADA, Pascal, Modula-2 or C, to name just a few) had language support for modules: you could define procedures as belonging to a module, could *export* certain procedures that would then be usable by other modules that *imported* these procedures. Such language support for modularisation increased reuse. Writing libraries of code (for mathematical functions, file handling, widget sets, ...) was possible because of this language support, and was widely used.

But, as we all know, this was not the end of the story. One of the problems faced by the modularisation approach was *customizability*. While modularisation made it possible to chop an application up in several pieces that you could develop and test more independently, it was very hard to customize them. For example, calling a function from a mathematical library is easy, but it's very hard to customize when a slightly different one is needed. The reason is that the only

composition mechanism that is supported is (a form of) delegation. Hence, solutions were sought to address such (and related) problems.

Object-oriented programming came along touting that it was going to solve this reuse problem. To attain this goal, it featured new composition mechanisms, such as *aggregation* and *inheritance*. As a result, you could not only call methods on objects in a procedural manner, but you could also make subclasses and aggregations. This made it more easy to reuse (large parts of) existing code without rewriting it. The most interesting aspect was probably the dynamic aspect: the fact that the receiver of a message could change at runtime. This 'late of binding' of self makes polymorphism a very powerful tool to attain better customizability.

When the dust in the object-oriented community more or less settled, class-based languages were the ones that left standing, and they are still used in the majority of object-oriented languages. But, more importantly, people had discovered another advantage of inheritance: it allowed not only *code reuse* but also, and more importantly, *reuse of design*. Frameworks, with their focus on the *Hollywood principle*, are the pinnacle of this evolution. An object-oriented framework is defined as a set of classes which embody an abstract design for solutions to a family of related problems [7]. It can be seen as a skeleton that implements an abstract application for some specific domain. Users of the framework customize by adding subclasses that implement or change the behaviour of the system. Hence, the overall design and functionality of the framework is used all over again, and customizations can be made to target smaller issues. While frameworks are indeed nice to use, they also face severe problems that hinder their acceptance:

1. writing a framework is very hard. As it implements an abstract solution to a whole range of similar problems, the good abstractions have to be implemented. When it is too abstract, people still have to write too much code. When it is not abstract enough, people have to work around the framework to target it to their issues. Only after several iterations and when applied in several contexts can it mature enough to be really useful.
2. understanding a framework is also very hard. The reason is that it is typically a big system that one wants to customize in a certain place, if possible without needing to browse the whole code in order to know which methods to override from which classes.
3. it is also very hard to document. While you can document its overall design, users are interested in very detailed information, that changes depending on the context they want to use it in.

The problem that lies at the core of these issues is that object-oriented programming is about *white-box reuse*: you need to know and understand the code before you can make a successful subclass. Consider for example the following well-known example: suppose we want to make a subclass *CountingSet* of a given class *Set* that keeps how many elements have been added. *Set* implements two methods: *add* for adding one element and *addAll* for adding all elements from another set. Then our subclass *CountingSet* needs to know how *Set* implements these operations. If the implementation of *addAll* in *Set* calls its *add* method, then *CountingSet* only has to override *add* to count the additions. If the implementation of *addAll* does not use *add* but immediately adds all methods, then *CountingSet* needs to override both methods.

Several approaches were taken to alleviate these problems and provide more constrained forms of white-box reuse (such as specialization interfaces [11] or reuse contracts [12]. The design pattern movement aimed at providing a better way of documenting, communicating and reusing solutions to common design problems [8, 5], like the (active) cookbook approach before it [10]. Other composition mechanisms were investigated, such as Aspect-Oriented Programming [9], Subject Oriented Programming [6], Generative Programming [4] and multi-dimensional separation of concerns [15]. However, despite all these important contributions and generally speaking, object-orientation did not completely live up to its promise to deliver general reusability.

3. FROM OBJECT TO COMPONENTS

Component software engineering now promises to 'deliver reusable, of-the shelf software components for incorporation into large applications' [14], and hence as the next general solution for the reuse problem. The basic idea of components is that of *black-box reuse* applied on wrapped binary components. Black-box reuse is considered better for reuse because it completely hides the internals of the component, and to reuse the component people do not need to know those internals to compose components. So, components are typically defined as entities that encapsulate some internal representation with one or more interfaces. Other components can invoke functionality through these interfaces. From the reuse point of view, this provides some advantages. First of all, this allows binary composition of components, something where the white-box (source code) reuse of object-oriented programming has no language support for. Also, components can be implemented in different implementation languages (as long as they share a compatible interface and runtime mechanism). Or, in a number of cases, it becomes easier to support distribution of components in an easier and more transparent way.

But, by currently restricting components to black-box reuse and delegation, component-oriented development risks in making the same pitfall as module-based approaches. Just as with modularisation, the composition mechanism supported is delegation, and customizing part of a component is very hard. Note that we are not claiming that white-box reuse is better; time has proved that this is too much at the other end of the spectrum. But we are claiming that, in order to make reuse work, language designers will have to look carefully at composition mechanisms for black-box components that go beyond simple delegation. We firmly believe that this should be one of the major conceptual contributions to make by component languages.

4. FROM COMPONENTS TO PURE COMPONENT LANGUAGES

Even though at the moment several component systems exist on top of object-oriented programming languages, we feel that there is need for pure component languages. These languages are needed to provide a component developer with a clean and concise vocabulary and semantics for building and composing components. We feel that, just as object-oriented programming languages gave rise to the notion of 'reuse of design', better component languages will answer new problems (and raise new questions).

At this moment we do not claim that we have a full solution for a pure component language. However, what we want to discuss is a fundamental issue to reusability (and hence component languages): composition mechanisms. We feel that current component approaches focus too much on only one possible composition mechanism: 'swappability', i.e. the fact that one component can be exchanged with another one. While this is certainly a very important issue, it's not the only one. Let's reconsider our toy example with *Set* and *CountingSet* again. The goal of making the subclass is to reuse as much of Set as possible, without having to specify everything anew. In a component language that only supports 'swappability' we have to use a delegation approach, meaning that we might end up implementing a lot of code that just delegates behaviour from *CountingSet* to *Set*. A possible answer might be another composition mechanism (a form of black-box inheritance).

So, we feel that composition mechanisms for black-box components should be the major focus for pure component languages. Concretely we propose to have a difference between the language in which components are implemented, and the composition languages to compose them. Even more, we foresee different composition languages depending on where the components are used. While in object-oriented programming languages the composition mechanisms are fixed by the language, we feel that there should be more flexibility, especially at this stage of the component language design.

5. EXAMPLES OF POSSIBLE COMPOSITION LANGUAGES

Since we think composition languages should be the major focus (and contribution) for component languages, we want to give some concrete examples. More specifically, we want to show how a logic or a functional language could be used as composition languages. These languages are not full component languages as such, but they can serve as a test-platform to experiment with some of the ideas.

5.1 QSoul: a logic approach

The first approach we would like to introduce is to use a logic programming language to compose components. The fundamental motivation for using a logic programming language is that composition of components has to do with expressing relations between these components, and logic programming deals with expressing and solving relations. Thus different composition mechanisms can be expressed as logic queries over components, and these queries determine the result of the overall composition.

More concretely, we think that a hybrid logic language such as *QSoul* is a natural candidate. QSoul is a logic programming language that features reflection with its implementation language [17, 16]. It allows full logic reasoning over objects and their structural aspects. We have used this to extract structural information from class hierarchies and to support round-trip engineering activities. More recently, QSoul is used as composition language for components for embedded devices (in the same way as it was used as an Architectural Description Language before) [18]. In this setup:

1. components are implemented with objects; they have a name, lists of properties (key-value pairs), lists of ports and possibly subcomponents. Since they are objects, these components can also be wrappers around other implementations (components in another language or values from a database, for example);

2. connectors connect the ports of components. These connectors are also implemented as objects that connect ports;

3. components can have consistency rules; these are logic rules that express certain constraints on the internals of the component. They can refer to the properties, ports and possible subcomponents of a component. An example of such a rule is that *a component that has a 'schedulable' property should also have a 'cycletime' property*;

4. composition rules express the semantics of the connector, and hence of the component composition. For example, these logic rules express that a connector can only connect two parts that are type-compatible, and what type-compatibility means. We are currently adding connectors that express inheritance for components, and will afterwards look at composite ports;

5. the overall architecture of the system consists of bundles of logic rules that express the consistency rules and composition rules available in this component language.

The nice thing of using a hybrid language such as QSoul is that it allows to express components and connectors as objects (data) and the composition mechanism (relations) in separated languages. It is easy to image a pure component language that uses a similar mechanism to separate the components from the compositions. With respect to current languages, it allows one to add other composition mechanisms than the ones provided 'out of the box'. Hence, 'swappability' of components is currently defined as components that have the same interfaces, but can also be defined type-compatibility of components. Inheritance, on the other hand, can be defined as well, with precise semantics. We can also look at adaptors that adapt interfaces of components to make them compatible. Hence we expect to use QSoul as an experimental vehicle for looking at several composition approaches, and using them on concrete examples.

5.2 Piccola: a functional approach

Piccola is an experimental language for composing applications from software components [2, 1]. Piccola is defined by a thin layer of syntactic sugar on top of a semantic core

based on Milner's pi calculus. (Piccola stands for PI Calculus based COmposition LAnguage.). Piccola is designed to make it easy to define high-level connectors for composing and coordinating software components written in other languages. It goes beyond scripting languages because it is not biased towards one particular scripting paradigm. Instead, it allows components to be posed according to different compositional styles. As such, Piccola focuses on providing mechanisms for building different kinds of compositional abstractions, namely wrappers, adapters, connectors, coordination abstractions, and generic glue code.

In Piccola, everything is a "form", a kind of immutable, extensible record that is useful for modelling objects, components, configurations, communications, default values and namespaces. Currently scripts exist that define and use different compositional styles such as aggregation, functional composition, inheritance, mixin composition, stream composition, ... At this moment, Piccola can be used standalone (meaning that components are implemented as Piccola forms), but forms can also wrap Java or Smalltalk objects. In that case, Piccola is used as a high-level scripting languages to compose forms wrapping objects.

6. CONCLUSION

We feel that it is time for pure Component Languages to support component-oriented programming. However, just as with the start of object-oriented programming several years ago, it is currently hard to see all the topics and possible choices, and several debates and 'language wars' will be held the coming years. At the end, a consensus will grow around what a component language should be. We feel that this workshop is an excellent discussion forum to address this topic, and think we can participate to provide some answers based on our extensive experience in designing and combining languages. More specifically, we feel that one of the important problems that should be addressed by composition languages is the aspect of composition of components. More specifically, we propose component languages where components are black-box abstractions, and with (one or more) composition languages to glue them together. As an example we show a functional (Piccola) and a logic (QSoul) composition approach.

7. REFERENCES

[1] F. Achermann, M. Lumpe, J.-G. Schneider, and O. Nierstrasz. Piccola – a small composition language. In H. Bowman and J. Derrick., editors, *Formal Methods for Distributed Processing, an Object Oriented Approach*. Cambridge University Press., 2001. to appear.

[2] F. Achermann and O. Nierstrasz. Applications = Components + Scripts – A Tour of Piccola. In M. Aksit, editor, *Software Architectures and Component Technology*. Kluwer, 2001. to appear.

[3] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP'90, ACM SIGPLAN Notices*, pages 303–311, Oct. 1990. Published as Proceedings OOPSLA/ECOOP'90, ACM SIGPLAN Notices, volume 25, number 10.

[4] K. Czarnecki and U. Eisenecker. *Generative Programming. Methods, Tools, and Applications*. Addison-Wesley, 2000.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addisson-Wesley, 1994.

[6] W. Harrison and H. Ossher. Subject-oriented programming (A critique of pure objects). In A. Paepcke, editor, *OOPSLA 1993 Conference Proceedings*, volume 28 of *ACM SIGPLAN Notices*, pages 411–428. ACM Press, Oct. 1993.

[7] R. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June 1988.

[8] R. E. Johnson. Documenting frameworks using patterns. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, volume 27-10, pages 63–76, 1992.

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of ECOOP'97*, pages 220–242. Springer Verlag, 1997. LNCS 1241.

[10] G. E. Krasner and S. T. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. Technical report, Xerox, Palo Alto, 1988. IB-D913170.

[11] J. Lamping. Typing the specialization interface. In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, pages 201–214. ACM Press, 1993.

[12] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse contracts: Managing the evolution of reusable assets. In *Proceedings OOPSLA '96, ACM SIGPLAN Notices*, pages 268–285. ACM Press, 1996.

[13] P. Steyaert and W. D. Meuter. A marriage of class- and object-based inheritance without unwanted children. In W. Olthoff, editor, *Proceedings ECOOP'95*, LNCS 952, pages 127–144, Aarhus, Denmark, Aug. 1995. Springer-Verlag.

[14] C. A. Szyperski. *Component Software*. Addison-Wesley, 1998.

[15] P. Tarr, H. Ossher, W. Harrison, and S. 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.

[16] R. Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001.

[17] R. Wuyts and S. Ducasse. Symbiotic reflection between an object-oriented and a logic programming language. In *Multiparadigm Programming with Object-Oriented Languages*, volume 7, pages 81–96. John von Neumann Institute for Computing, 2001.

[18] R. Wuyts, S. Ducasse, and G. Arévalo. Applying experiences with declarative codifications of software architectures on cod. In *6th Workshop on Component Oriented Programming (ECOOP'01)*, 2001.