First-Class Compositions
Defining and Composing Object and Aspect Compositions with First-Class Operators

Lodewijk Bergmans, Wilke Havinga, and Mehmet Aksit
University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands
{l.m.j.bergmans,w.havinga,m.aksit}@ewi.utwente.nl

Abstract. A considerable amount of research, especially within the OO and AOSD communities, has focused on understanding the potential and limitations of various composition techniques. This has led to a large number of proposals for alternative composition techniques, including many variations of message dispatch, inheritance, and aspect mechanisms. This paper makes the case that there is no single perfect composition technique that suits every situation, since different techniques incur different tradeoffs. The proper composition technique to use depends on the particular design problem and its requirements (such as the required adaptability, reusability, understandability and robustness). However, most programming languages limit the available composition techniques to a very few. To address this, we propose a novel composition model, called Co-op. The model provides dedicated abstractions that can be used to express a wide variety of object composition techniques (“composition operators”). Examples include various forms of inheritance, delegation, and aspects. The proposed model unifies objects (with encapsulated state and a message interface) and composition operators; composition operators are specified as first-class citizens. Multiple composition operators can be combined within the same application, and composition operators can even be used to compose new composition operators from existing ones. This opens new possibilities for developing domain-specific composition operators, taxonomies of composition operators, and for reuse and refinement of composition operators. To validate and experiment with the proposed model, we have designed and implemented a simple language, Co-op/I, that we also use in this paper to show concrete examples.

1 Introduction

The history of programming languages shows a continuous search for new—presumably better—composition techniques. A typical aim of such techniques is to find better ways for structuring increasingly complex software systems into modules that can be developed and reused independently.

Composition operators are language mechanisms that let programmers define part of the behavior and/or data of an abstraction, in terms of the behavior
and/or data of one or more other abstractions, by means of a composition expression. A typical example of a composition operator is inheritance. Inheritance takes the definition of a new (sub)class, and combines its behavior with that of an existing (super)class. Other examples of composition operators include delegation, pointcut-advice mechanisms, composition filters, mixins, traits, etc.

Most languages adopt a fixed set of composition operators, typically with explicit notations and predefined semantics. In case a language does not provide a composition operator with the desired compositional behavior, programmers may need to write workarounds in their applications; by adding glue code, or by using macros, libraries, frameworks, or language extensions. However, typically, such workarounds are not integrated with the language, and the resulting abstractions suffer from lack of comprehensibility, adaptability, and reusability.

The availability of only a limited set of composition operators causes additional issues; most existing languages have a bias toward one kind of decomposition of software systems, which also imposes constraints on the viability of particular evolution scenarios, or in other words, the extensibility of software. Thus, each composition operator (and hence, language) has a bias that makes some types of evolution scenarios easier to accommodate, or less error-prone, than others. Such trade-offs are inherent to the choice of particular composition operators—there exists no single composition operator that is able to address all kinds of evolution scenarios equally well, while still providing meaningful higher-level abstractions.

To work toward addressing the issues identified above, we present a composition infrastructure that (a) supports the definition of a range of composition mechanisms, (b) allows composition mechanisms to be expressed in terms of first-class entities, enabling the construction of new composition mechanisms from existing ones, (c) supports the use of multiple composition mechanisms within the same program, while (d) supporting a variety of aspect- as well as object-based composition mechanisms.

A language construct is defined to be a first-class entity in that language when there are no restrictions on how it can be created and used; when the construct can be treated as a value without restrictions. Finkel defines a language construct to be a first-class value if the value can be passed as a parameter, returned from a procedure, and assigned to a variable. Others include the ability to construct the value at runtime (cf. instantiation), and require an intrinsic identity.

Our approach has been implemented and is presented in this paper in terms of a small language, called “Co-op/I”. In this language, composition operators can be constructed using several “primitive” elements, such as selectors, bindings, actions and constraints, which can be used to define composition operators. These primitive elements are expressed in terms of first-class entities (objects), so that they can be freely composed. We use these primitive entities to express several composition mechanisms, including different styles of inheritance, as found for example in Smalltalk or Beta, as well as aspects.

1 This is similar to the “tyranny of the dominant decomposition”.

First-Class Compositions