Free Composition Instead of Language Dictatorship

Lodewijk Bergmans, Steven te Brinke, Christoph Bockisch and Mehmet Akşit
University of Twente, Enschede, The Netherlands

Keywords: Language Design, Language Engineering, Software Composition, Free Composition.

Abstract: Historically, programming languages have been—benevolent—dictators: reducing all possible semantics to specific ones offered by a few built-in language constructs. Over the years, some programming languages have freed the programmers from the restrictions to use only built-in libraries, built-in data types, and built-in type-checking rules. Even though—arguably—such freedom could lead to anarchy, or people shooting themselves in the foot, the contrary tends to be the case: a language that does not allow for extensibility is depriving software engineers of the ability to construct proper abstractions and to structure software in the most optimal way. Therefore the software becomes less structured and maintainable than would be possible if the software engineer could express the behavior of the program with the most appropriate abstractions. The idea proposed by this paper is to move composition from built-in language constructs to programmable, first-class abstractions in a language. We discuss several prototypes of the Co-op language, which show that it is possible, with a relatively simple model, to express a wide range of compositions as first-class concepts.

1 MOTIVATION

The software engineering discipline faces many challenges; one of the important challenges is to cope with the changes that need to be incorporated into software systems during their lifetime. A particular difficulty is that small, local changes in the requirements of a system, often lead to non-local changes in the software. This is caused by the fact that the structure of the implementation tends to be significantly different from the structure of the problem domain.

Another software engineering challenge is managing the complexity of software (Royce, 2009). We are building increasingly large software systems. Such systems encompass a substantial amount of *inherent complexity*; partially in the problem domain and partially in the solution domain. However, the *realization* of these systems also introduces a large amount of *accidental complexity* (Brooks, 1987), while going from the conceptual solution to a realization model.

A key argument adopted in this paper, and previously coined, e.g., by Brooks (Brooks, 1987), is that the limited ability of realization models to accurately represent the concepts and their interdependencies in a conceptual solution is the main cause of *accidental complexity*. As a result, the complexity of realizations is typically substantially larger than the complexity of the conceptual solution.

In software engineering, a key strategy for dealing with change and managing complexity is "divide and conquer": achieve *separation of concerns* (Dijkstra, 1976) by dividing a solution into building blocks and delivering working systems by expressing proper compositions of these building blocks. The history of programming and design shows a steady movement towards supporting higher-level abstractions of building blocks and more advanced ways of expressing such compositions. For example, object-oriented and aspect-oriented programming are largely motivated by the need for improved modularity and separation of concerns; recent trends in software engineering, such as Model-Driven Engineering (MDE) and Domain-Specific Languages (DSLs), all aim at offering an appropriate abstraction level for expressing particular types of problems: to this extent, they offer (a) dedicated (possibly graphical) syntax, (b) dedicated data types and operators, or (c) dedicated abstractions and corresponding composition techniques, to achieve better modularity and separation of concerns for specific domains.

The message of this paper is that there is substantial benefit in offering languages where abstractions and composition techniques are not hard-wired; rather they should be easy to introduce on demand by software engineers when new kinds of compositions are identified.

Bergmans L., te Brinke S., Bockisch C. and Akşit M..
Free Composition Instead of Language Dictatorship.
DOI: 10.5220/0004082603880393
In *Proceedings of the 7th International Conference on Software Paradigm Trends* (ICSOFT-2012), pages 388-393
ISBN: 978-989-8565-19-8

This paper is an extension of a paper previously presented at the workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE) 2011. Sections 6.2 and 6.3 summarize our publications in the workshop on Free Composition (FREECO) at ECOOP 2011 (Te Brinke et al., 2011a) respectively at Onward! 2011 (Te Brinke et al., 2011b).

2 COMPOSITION MECHANISMS

Programming languages[1] offer explicit and implicit means to express composition of abstractions, which means that the characteristics of a new abstraction are expressed in terms of the characteristics of one or more existing abstractions (and possibly additional specifications). For example, *function composition* in functional programming, such as $f(g())$, expresses that—the behavior of—f and g are composed by using the result of g as an argument of f. Similarly, *function invocation*, such as the call to q in $p()\{p_1;q();p_2\}$ is used to define part of the behavior of a procedure p in terms of the behavior of q. In this sense, the function invocation expresses that the behavior of p is a composition of the behavior of p_1, $q()$ and p_2. Another example is *object aggregation* (also referred to as *object composition*): this expresses how a new object structure is defined as, e.g., the union of other object structures.

In typical object-oriented languages, the following composition mechanisms are available: behavior composition through message passing (cf. function invocation), object aggregation, and inheritance. Compositions can be binary (such as function invocation and single inheritance) or n-ary (such as multiple inheritance or pointcut-advice composition in AOP).

Composition is not necessarily implemented *directly* by language constructs; for example, many design patterns describe how a set of objects interact to create a coherent new behavior: a composition of the participating objects. This composition is typically realized by a set of code snippets distributed over the participating objects, which must be re-implemented for every design pattern instantiation, it affects the traceability of the patterns in the code, and is hard to maintain.

[1]Whenever we talk about languages in this paper, this should be interpreted broadly to any means of specifying machine-executable behavior.

3 PROBLEM ANALYSIS

New composition mechanisms are introduced all the time. For example, Taivalsaari (Taivalsaari, 1996) describes a taxonomy for inheritance mechanisms, from which—in theory—hundreds of variants for inheritance can be derived. More recently, many proposals have been made for aspect-oriented languages and models: a survey report (Brichau et al., 2005) contains 45 different proposals, where in most cases the composition techniques are unique. Similarly, there is an indefinite amount of design patterns that essentially express a composition of several objects.

In general, it can be safely assumed that for each of these proposed composition mechanisms, there is a sound argumentation why that mechanism works better—at least for a certain class of applications, or within a certain context. The fundamental reason is that the application of each composition mechanism involves a certain trade-off, which makes it particularly suitable in certain contexts, but less so in others. Hence, a language that dictates a fixed set of composition techniques, with no opportunity to extend that set, will inherently restrain the software engineer: he or she is not able to choose the most appropriate composition mechanism, with the best possible trade-offs, and the most natural mapping from a conceptual solution to the implementation. Among the negative results caused by such dictatorship[2] are:

Lack of Traceability since the intended composition has to be replaced with an alternative, typically involving additional 'glue code'.

Lack of Maintainability because the glue code is usually specific to the context, and has to be added in multiple locations, where it is also tangled with the functionality.

Increased Accidental Complexity because straightforward compositions at the conceptual level have to be realized by more complicated code that introduces additional dependencies.

4 SCOPE AND ASSUMPTIONS

By now the strategy that we propose in this paper may be obvious; we want to free composition from languages where it is limited to a few composition mechanisms, and instead propose language facilities where the appropriate mechanisms can be defined, applied, and reused. Before explaining this approach in more

[2]There are in fact also positive sides; e.g. less choice makes both decisions and comprehension easier—albeit at substantial costs.

detail, we first discuss the scope of our solution approach and some of the assumptions we make. This discussion should also help to distinguish our work from various areas of related work.

Although not essential to the general idea, in the remainder of this paper we focus on object-based languages, which support encapsulation and message-invocation. In this context, composition refers to *composition of the behavior of objects*.

Our approach is *not* based on full reflection: although that is a very powerful technique, reflection-based solutions tend to be very hard to organize and manage and, hence, are not suitable from a scalability point-of-view. In particular, building fully reflective solutions that are also composable and extensible, requires a substantial amount of effort and discipline for the involved software engineers. However, we do propose to adopt limited reflection (where only specific elements of a language are exposed for reflection). We would like to point out that our approach can well be implemented, *using* reflection, as a specific Meta-Object protocol (MOP) (Kiczales et al., 1991).

We also aim for solutions that are not transformation-based, such as most Model-Driven Engineering (MDE) tools and external Domain-Specific Languages (DSLs). Although implementing composition operators as transformations is in principle a possibility, this suffers from several issues. In particular, if multiple composition operators are to be integrated within a larger solution: (1) it is hard to exchange data between the model (or DSL) world and the rest of the system, (2) it is hard to add additional DSLs without running into all kinds of integration issues, and (3) it requires additional tool support to develop at the model level.

5 GENERAL APPROACH

In the general approach of *first-class composition operators* (Co-op), composition operators are objects which *operate* on compositions. Composition operators can (cooperatively) influence the behavior of a composition. As we will discuss, this approach is sufficiently powerful to express common composition mechanisms such as inheritance, delegation, and design patterns. Key characteristics of a language adopting the Co-op approach are the following:

- Only one primitive composition operator is provided by the language, which is *sending a message*. Other composition operators such as inheritance are not manifested in the language's syntax.
- New composition operators can be defined and added to a system (a key characteristic of Co-op).
- Composition operators are first-class entities in the language: we believe this is an essential feature for a scalable approach (see also composability (3) below). This also means that the application of composition operators can be as simple as object instantiation, and composition operators can be managed as regular libraries.
- Composability (1): it is possible to freely apply multiple composition operators within the same application.
- Composability (2): it is *possible* to combine multiple composition operators in the same context, i.e. apply them to the same objects. Nevertheless, in some cases the semantics of those operators may be such that a combined usage is not desirable.
- Composability (3): composition operators can be used in the definition of other composition operators (as long as this does not cause infinite recursion).
- Inherent dependencies or constraints among composition operators are expressible, such as exclusion, relative ordering, and overriding among composition operators.

While the syntax for a *message send* is embodied in a language that follows our approach, the semantics of where a message is delivered is not fixed, but determined by the composition operators in Co-op: Composition is performed at message sends, by rewriting the message's properties. A message send is, by itself, undirected; i.e., when the program sends a message, it specifies property values such as the message name or initial argument values; since all message properties may be rewritten by composition operators, e.g., the first argument value is not necessarily the receiver of the message.

Figure 1 shows the composition infrastructure schematically. In *Co-op*, we have chosen to use the terms *module* and *module instance* to refer to concepts that are comparable to (object-based) classes and objects, with the distinction that they may also specify composition operators. In the middle left, a *module instance* (object) performs a *message send*. The reified message is evaluated by the active set of *bindings*, defined by *Co-op modules*. Finally, the evaluation of a message send typically leads to the invocation of one (or more) operation(s) on a module instance. The key characteristic is that all possible behavior involves message dispatch; hence manipulating message dispatch can be used to express a truly wide range of behavioral compositions.

Composition operators are modules (classes) that can specify one or more of the following three additional members to define how messages are eventually

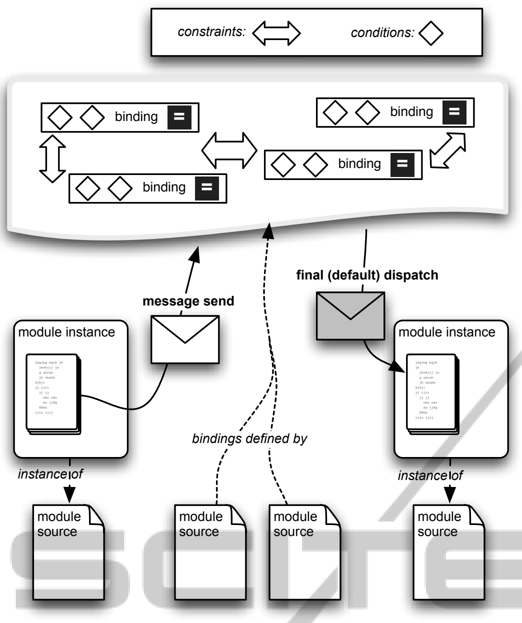

Figure 1: Overview: composition in *Co-op*.

bound to concrete operations; this in turn expresses the semantics of a particular composition technique:

1. *Conditions* define boolean predicates that refer to message properties and to fields or methods defined in the class.
2. *Bindings* define the conditions to specify which messages are selected to participate in the rewriting of a message, and assign new values to message properties when they are applicable; hence they bind source messages to new target messages. The target messages may be directly executable, or may be manipulated once more by other bindings.
3. *Constraints* define relations between composition operators with bindings that are applicable at the same message send. Relations are, for example, the order of evaluation and the prevention of evaluating one operator. This can express exclusion, ordering, and overriding of compositions.

6 EVIDENCE: IMPLEMENTING THE CO-OP IDEA IN A LANGUAGE

As a proof-of-concept, we have partially realized these ideas in three language implementations of *Co-op* as object-oriented, dynamically typed languages.[3]

[3]See the SourceForge project *Co-op* at co-op.sf.net.

In this section, we will present these three implementations.

6.1 The Co-op/I Language

For Co-op/I (Bergmans et al., 2011; Havinga et al., 2010a), we have defined an operational semantics and implemented an interpreter for it in Java. Co-op/I faithfully realizes the execution approach described in the previous section and exposes all method invocations as *message sends*.

In this prototype language we have been able to realize many different composition operators such as aspects, delegation, traits, different forms of inheritance (Havinga et al., 2010a), and several design patterns (including the Memoization, Observer and State patterns (Havinga et al., 2010b)). Of the different alternative semantics for inheritance presented by Taivalsaari (Taivalsaari, 1996), we have implemented all, except for those with ordered multiple inheritance, due to a limitation in the Co-op/I implementation, not in the concept. In our examples, we have been able to reuse the implementation of lower-level composition operators by combining them to more complex ones, again by means of Co-op/I composition operators.

6.2 The Co-op/II Language

As a successor of Co-op/I, we designed a second prototype of a Co-op language and execution environment, Co-op/II (Te Brinke et al., 2011a). In this prototype, additional to function calls, also data accesses are reified as messages being sent and composition operators can reason about and influence such messages.

Composition techniques such as inheritance do not only control the composition of behavior, but also the composition of data. For example, access modifiers control from where data fields can be accessed: e.g., only from methods defined in the same class as the field declaration or also from methods defined in classes inheriting from the class declaring the field.

Most (OO) programming languages have built-in language constructs to manipulate the way that data is—or can be—accessed. A few examples are:

- Access modifiers in Java, C++ and C# are *public*, *protected*, and *private*. A language like C++ adds a *friend* keyword to express yet another form of access rights on data (as well as behavior). Note that there is a wide range of possible access modifiers, when including the notion of package-level protection, or the distinction between class-level and instance-level protection.

- The Java, C++, and C# keyword *static* controls whether all instances of a class share a field, or each has its own copy.
- The keywords *final* in Java, *const* in C++, and *readonly* in C# declare special semantics to the usage of a variable (i.e., the variable may be assigned only once).

In general, examples of modified composition semantics, which Co-op/II facilitates in addition to those supported already by Co-op/I, are: access modifiers (static, synchronized, final, and so forth) and also more conceptual constructs such as automatic conversions, checking of validity constraints, persistence, transactions, and expressing roles.

Co-op/II provides a more declarative language for defining composition operators than Co-op/I. This allows reasoning about composition operators to, for example, provide optimizations and check their correctness. Identifying the precise amount of reasoning that can be done is still future work.

6.3 The Co-op/III Language

At the moment, we are working on yet another Co-op implementation, Co-op/III (Te Brinke et al., 2011b). In addition to the request-reply model of method invocation and data access, supported already by Co-op/I and Co-op/II, this language has a focus on control-flow-related semantics of compositions. By this, we mean inter-procedural constructs such as exception handling, co-routines, and around advice with proceed in AOP, as well as intra-procedural control structures such as conditionals and loops.

This will allow defining, e.g., different semantics for exception handling such as adding support to retry—or even resume—the operation that caused an exception. At the same time, e.g., loop constructs with different semantics can be defined in Co-op/III, such as evaluating the loop condition before, after or even during the loop (Dahl et al., 1972).

The current implementation of Co-op/III will also unify operators (such as addition, subtraction, and string concatenation) with method invocations. Thus, all operators are message sends that can be influenced by the programmer.

6.4 Discussion

While our implementations of the Co-op languages demonstrate that it is in principle possible to make a language sufficiently powerful to define at least a wide range of composition mechanisms, some questions remain. For example, the following issues must be addressed to make this approach ready for practical use: the language must be made more complete to support, e.g., ordered multiple inheritance; the execution performance of Co-op programs must be optimized; and inter-operation with other programming languages should be supported.

Optimizations are necessary because the dynamic evaluation of composition operators adds an overhead to each message send in the program. To compensate, the Co-op/II and Co-op/III prototypes already aim at a high degree of declarativity in the definition of conditions, bindings and constraints. Thus, their effects on a message can be partially evaluated before runtime. However, it is still an open topic to actually implement and evaluate such optimizations for Co-op.

Language interoperability is important to use the large body of existing libraries. This is supported by main-stream languages, e.g., the Java Native Interface allows to use C/C++ libraries from Java and vice versa. To support such interoperability, ways for communicating between the dynamically-typed Co-op world and the statically-typed world of Java or C++ must be researched. A bridge to another dynamically typed language, e.g. Smalltalk, may be simpler, but still not trivial: also Smalltalk has a notion of *type* hierarchies and assignment compatibility while Co-op composes the *behavior*.

7 IMPLEMENTATION STRATEGIES

Facilities for message rewriting can also be offered in other ways, most obviously through a reflective language, as a meta-object protocol (MOP). As already explained in Section 4, the Co-op *model* and a MOP approach are not contradictory, and a MOP approach is one way to implement the Co-op ideas. However, the advantage of defining a language is that it can offer the programmer dedicated abstractions for, e.g., the conditions, bindings and constraints. This can make the specification more declarative, which makes composition operators, themselves, more composable. Such declarativeness furthermore enables analyses for better IDE support or performance optimization.

Various OO programming languages (e.g. SELF (Ungar and Smith, 1987) and Smalltalk (Goldberg and Robson, 1983)) aim at offering a simple core language on which to build complex abstractions. However, to the best of our knowledge, these languages either do not offer the ability to define tailored composition semantics or do so by opening up the language in a generic way through reflection, as discussed in the previous paragraph.

Another conceivable approach is to implement composition operators by means of code transformations. They can be implemented in terms of a compile-time MOP (Sheard and Jones, 2002), which means that the transformations can be defined in the same language as the application code. But still transformations cannot be freely composed because they, generally, make assumptions on the code structure; this may have been modified by a previously applied transformation.

8 CONCLUSIONS

The message of this paper is that, when implementing an application, dedicated abstractions that fit the problem domain are needed; and since programs, in general, combine multiple problem domains in unforeseeable ways, programming languages must not hardwire abstractions and composition mechanisms, but must allow developers to tailor them to their needs.

For this purpose, we propose the Co-op approach to integrate facilities into programming languages that allow developers to implement their own composition mechanisms according to their domain's abstractions. They should be implemented as composition operators which are first class objects in the programming language such that they can be composed, themselves, by means of custom composition operators. By discussing the Co-op language prototypes that allow normal application objects to act as composition operators by means of message rewriting, we have shown that realizing these ideas is feasible.

While the Co-op approach—allowing programmers to define their own composition operators as first-class objects—is quite powerful, it also bears more complexity. This complexity needs *not* be exposed to the day-to-day programmer, though, because the implementation of composition operators can be provided by a few, well-trained experts through libraries. The common programmer simply uses such a library, instantiating composition operators as needed. Nevertheless, it is subject to future studies to research the impact of our proposed programming model in practice.

ACKNOWLEDGEMENTS

We would like to thank Wilke Havinga and Havva Gülay Gürbüz for their contributions to *Co-op*. This research work was partially funded through the ENOFES project (STW 11850).

REFERENCES

Bergmans, L. M. J., Havinga, W. K., and Akşit, M. (2011). First-class compositions–defining and composing object and aspect compositions with first-class operators. *Transactions on Aspect-Oriented Software Development*.

Brichau, J., Haupt, M., Leidenfrost, N., Rashid, A., Bergmans, L., et al. (2005). Report describing survey of aspect languages and models. Technical Report AOSD-Europe Deliverable D12, AOSD-Europe-VUB-01, Vrije Universiteit Brussel.

Te Brinke, S., Bergmans, L. M. J., and Bockisch, C. M. (2011a). The keyword revolution: Promoting language constructs for data access to first class citizens. In *Proc. 1st Int. Workshop on Free Composition*, FREECO '11, New York, NY, USA. ACM.

Te Brinke, S., Bockisch, C. M., and Bergmans, L. M. J. (2011b). Reuse of continuation-based control-flow abstractions. In *Proc. 2nd Workshop on Free Composition @ Onward! 2011*, FREECO-Onward! '11, pages 13–18, New York, NY, USA. ACM.

Brooks, F. (1987). No silver bullet: Essence and accidents of software engineering. *IEEE computer*, 20(4):10–19.

Dahl, O. J., Dijkstra, E. W., and Hoare, C. A. R., editors (1972). *Structured Programming*. Academic Press Ltd., London, UK.

Dijkstra, E. W. (1976). *A discipline of programming*. Prentice-Hall, Englewood Cliffs, New Jersey.

Goldberg, A. and Robson, D. (1983). *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

Havinga, W. K., Bergmans, L. M. J., and Akşit, M. (2010a). A model for composable composition operators: Expressing object and aspect compositions with first-class operators. In *Proc. 9th Int. Conf. on Aspect-Oriented Software Development*, pages 145–156, New York. ACM.

Havinga, W. K., Bockisch, C. M., and Bergmans, L. M. J. (2010b). A case for custom, composable composition operators. In *Proc. 1st Int. Workshop on Composition: Objects, Aspects, Components, Services and Product Lines*, volume 564 of *Workshop Proceedings*, pages 45–50. CEUR-WS.

Kiczales, G., des Rivieres, J., and Bobrow, D. G. (1991). *The Art of the Metaobject Protocol*. MIT Press, Cambridge, Massachusetts.

Royce, W. (2009). Improving software economics-top 10 principles of achieving agility at scale. White paper, IBM Rational.

Sheard, T. and Jones, S. P. (2002). Template meta-programming for Haskell. *SIGPLAN Not.*, 37:60–75.

Taivalsaari, A. (1996). On the notion of inheritance. *ACM Comput. Surv.*, 28(3):438–479.

Ungar, D. and Smith, R. B. (1987). Self: The power of simplicity. *SIGPLAN Not.*, 22:227–242.