On Composing Separated Concerns: Composability & Composition Anomalies

Lodewijk Bergmans
Bedir Tekinerdogan
Maurice Glandrup
Mehmet Aksit

TRESE Software Engineering group, Dept. of Computer Science, University of Twente,
P.O. Box 217, 7500 AE, Enschede, The Netherlands
+31-53-489 3767
{bergmans, bedir, glandrup, aksit}@cs.utwente.nl

ABSTRACT
It is generally acknowledged that separation of concerns is a key requirement for effective software engineering: it helps in managing the complexity of software and supports the maintainability of a software system. Separation of concerns makes only sense if the realizations in software of these concerns can be composed together effectively into a working program. The ability to compose systems from independently developed components that can be adapted and extended easily is a long-standing goal in the software engineering discipline. However, both research and practice have shown that composability of software is far from trivial and fails repeatedly. Typically this occurs when components exhibit complex behavior, in particular when multiple concerns are involved in a single component. We believe that, to address the composability problems, we need a better understanding of the requirements involved in successful composition, and in addition define the situations where composition fails. To this aim, in this paper we introduce a number of requirements for design-level composability and define a category of composition problems that are inherent for given composition models, which we term as *composition anomalies*.

1 Motivation and Overview

It is generally acknowledged that the notion of "separation of concerns" is one of the crucial issues in achieving adaptable, reusable and maintainable software. Separating concerns is a difficult design activity in itself, requiring extensive domain knowledge. After separating the concerns the next key issue is the composition of the implementations of the concerns.

One of the goals of this paper is to provide a better understanding of composability, so that better models and mechanism for composition of software can be designed. We adopt the term *composition scheme* to refer to conceptual models and mechanisms for composition in a computation model or programming language. Examples of well-known composition schemes are inheritance, aggregation and delegation.

The composition of software elements has been studied extensively throughout the history of computer science and many composition schemes have been proposed. In practice, however, still many composition problems occur. To reason about the deficiencies of these composition schemes we distinguish between the composition failures that are due to conceptual or design flaws and those that are due to the applied composition scheme. As such we can provide the following categories of composition problems:

1. Composition is not possible for logical reasons, in other words, one tries to compose components that are inherently not composable.
2. Composition cannot be realized because the adopted composition scheme does not support it, although composition is possible from the logical perspective.
3. Composition is realizable with the adopted composition scheme, but requires additional workarounds or glue code that reduces the maintainability of the resulting design.

In this paper we are primarily interested in the latter two cases, which are examples of the so-called *composition anomaly*. The remainder of this paper is organized as follows: In section 2 we describe the necessary requirements for composability and give a definition of composability. In section 3, we introduce the notion of composition anomaly, which designates the situations where concepts are logically composable but composition is not realizable due to the adopted composition schemes. Finally, in section 4 we conclude this paper.

2 Composability Requirements

We can state that the ease of composition of various concerns may vary in degree. Some concerns may be easily composed, others may need more effort, while again other concerns may not be composable at all. Composability is a quality requirement that designates 'the ability for

composition'. Both the concerns and the composition scheme that is used for the composition determine composability. For example, the composition of the concerns C1 and C2 to compose a new concern C3 using a composition scheme $\oplus$ can be expressed as follows:

$$C_3 = C_1 \oplus C_2.$$

Hereby the composition scheme $\oplus$ can be defined as a function $\oplus : C \times C \rightarrow C$. Note that many variations of such a composition scheme exist: e.g. extension of C_1 with C_2, composition from partial selections of C_1 and C_2, and the binding time may vary as well from compile time to run-time. Although we will discuss only the composition of two concerns C_1 and C_2, this can easily be generalized to the composition of any number of concerns.

It is obvious that the ease or difficulty of the composition depends both on the concerns and on the suitability of the composition scheme. In this section we investigate the properties that are inherent to (a given set of) concerns and determine the ability to compose concerns with each other, regardless of the concrete composition scheme to be used. We categorize the requirements for composing concerns into *functional composability* and *procedural composability*.

FUNCTIONAL COMPOSABILITY

Every composition must provide some function of value. Composition may be needed, for example, to enhance existing functionality or to define more complex behavior. Functional composability refers to the functionality of the overall composition. In order to achieve functional composability it is required that the composition is conceptually sound and provides useful and correct semantics:

- *Composition must be conceptually sound*:
  The composition of the concerns must be conceptually sound, or 'relevant'. This means that the overall composition must fulfill or support a predefined intention or goal. The goal is usually determined by a given need or problem.

  For example, the composition of a Buffer class with, say, a random number generator or an annuity algorithm does not make any sense. On the other hand, composing a Buffer with e.g. locking facilities, printer spooling or a telnet protocol implementation could be conceptually sound. Typically, relevance reveals itself through – possibly indirect– dependencies between the composed concerns (i.e. after composition).

- *Composition must have useful and correct semantics*:
  To support the predefined goal the composition must provide useful and correct semantics. The relevance of the semantics of the composition is dependent on the context and the requirements of the composition. The correctness of the semantics refers to the correct integration of the sub-concerns to provide the intended functionality of the overall composition.

  For example, assume that both a buffer and the locking concerns have been implemented in Java using the standard Java monitor-based synchronization scheme. This will typically be semantically incomposable, since the resulting nesting of monitors causes a deadlock in the standard Java library.

Procedural Composability

While functional composability refers to the requirements for the overall composition, procedural composability deals basically with the interoperability or the dependencies and interactions between components that are composed together. We may distinguish between three main levels of interoperability between components [Vallecilo 00]:

- *Signature level*, which refers to the names and signatures of the operations of the separate components
- *Protocol level*, which refers to the partial ordering of the operations and the blocking conditions.
- *Semantic level*, which refers to the semantics of the operations.

Combining Functional & Procedural Composability

Functional and procedural composability are largely orthogonal. Components may be procedural composable without being functionally composable. This means that an arbitrary set of components is composed successfully ('by accident') without providing any functionally meaningful behavior.

On the other hand, if components are functionally composable, but not procedurally, cooperation can only be obtained by introducing a suitable adaptation module, which translates between the interaction procedures used by the different components. The translation mechanism can be implemented as a wrapper around C_1 and/or C_2, as an abstraction which is responsible of converting operation names and/or attributes, or as an interpreter/compiler which translates calls between C_1 and C_2. It is also possible, that given fixed implementations of C_1 and C_2, no translation scheme can be defined that handles the procedural incomposabilities correctly.

Based on the above discussion, we provide the following definition for the term *composable*:

> Two concerns (or components implementing concerns) C_1 and C_2 are *composable* if (and only if):
> - C_1 and C_2 are *functional composable*, and
> - C_1 and C_2 are *procedural composable*

Note that the above are necessary conditions for a composition to succeed, but successful composition still depends on the concrete composition scheme that is adopted. However:

If two concerns C_1 and C_2 are *composable*, this means that there exists at least one composition scheme $\oplus$ that yields the composition $C_1 \oplus C_2 = C_3$.

Obviously, the fact that a composition scheme exists is a theoretical observation, one may not find a practical implementation of this composition scheme.

3 Composition Anomalies

An *anomaly* is a "deviation from the common rule; irregularity" or "something different, abnormal, peculiar, or not easily classified" (Merriam-Webster). With *composition anomaly* we refer to the cases where a composition that is intuitively and logically correct, is not realizable with a specific composition scheme. In other words, a composition of two concerns that are composable according to our definition in section 2 cannot be implemented (without loss of quality characteristics such as maintainability) with a particular composition scheme.

The term composition anomaly is a generalization of the term *inheritance anomaly*, as was coined by Matsuoka et.al. in [Matsuoka 90, 93] to denote the more specific case where the embedding of synchronization code in classes caused serious problems when trying to reuse and extend such code through inheritance mechanisms. In those cases, it typically appeared that the problems could be patched by overriding in a subclass substantial parts of the methods defined by a superclass.

Depending on the characteristics of the particular composition scheme, it may be possible, though, to fix a composition deficiency with some patching code. Although there is no fundamental problem with the need for additional code, it turns out that this reduces quality properties such as adaptability, reusability and maintainability, in virtually all cases. This may manifest itself e.g. by requiring code replication, or additional dependencies on either implementations or interfaces.

Definition

We will now define the meaning of *composition anomaly* in detail. Assume that C_1 and C_2 are *composable* according to our definition in section 2 (i.e. both functionally and procedurally composable). That is, the two concerns, C_1 and C_2, are (logically) composable through some composition scheme $\oplus$. Now assume that we try to realize the composition with another, given composition scheme $\oplus'$, which should yield the composed concern C_3.

We can distinguish the following results of the composition:

1. $C_1 \oplus' C_2 = C_3$; the composition is successful and yields the desired concern C_3. This means the composition scheme $\oplus'$ is suitable, no composition anomaly has occurred.
2. $C_1 \oplus' C_2 = \perp$; the composition is not possible and does not yield a composed result. This means the composition scheme $\oplus'$ is not suitable for this composition. This is considered as a composition anomaly, since we established that C_1 and C_2 *are* composable.
3. $C_1 \oplus' C_2 = C_3'$ (*where* $C_3' \neq C_3$) ; the composition is possible, but does not yield the desired C_3. This is obviously not the desired case, but is not always an example of inheritance anomaly, as we will discuss in the following.

The situation in the last case, a composition that results in a composed concern C_3', will (may) trigger a software engineer to try to make adaptations to resolve this situation. The following adaptations can be considered:

3.1 Adopt a different (instantiation of the) composition scheme to replace $\oplus'$; the ability to do so depends on the facilities available in the realization environment (typically the adopted implementation language). Then the procedure starts from the beginning.

3.2 Adapt any of the concerns C_1 or C_2 to solve the problem. This may be a feasible solution, but is in general out of the question; a reused class should never be modified to accommodate a specific reuse (composition) context [1]. If adaptation of C_1 or C_2 is the only way to achieve the desired composition, it is considered a composition anomaly.

3.3 Enhance C_3' with extra, so-called *glue code*, to achieve the desired (behavior of) C_3. A typical example of such glue code in the case of inheritance-based composition schemes is the redefinition (overriding) of methods to fix the missing or undesired behavior. Although this may indeed yield a desired C_3, the addition of glue code will in most cases have a negative impact on the quality of the code (design), in particular the maintainability. If this is the case, this is also considered an example of composition anomaly, since the composition did not yield the 'clean', maintainable, C_3 that was intended.

We can express the glue code as a function over the composition, yielding the desired C_3: $g(C_1 \oplus' C_2) = C_3$. We may again categorize the situation where glue code is added according to the following characteristics:

3.3a '*clean glue code*': this means that the addition of glue code does not have a negative impact on the quality, in particular the maintainability. This requires additions that are very modular and do not replicate or redefine existing code. We do not consider this case as a composition anomaly, but as successful composition.

3.3b *replicated code*: this typically occurs (a) when redefining an element from a reused concern (as in method overriding through inheritance), or (b) when a

[1] Unless we are dealing with refactoring or redesign of the system.

new element must have similar characteristics as an existing, reused, element. Since replicated code is destined to lead to maintenance problems, this case is considered undesired and as such an example of composition anomaly.

3.3c *n-ary (re-)definitions*: these occur when composition of two concepts involves crosscutting of implementation elements such as methods. Consider for example one bookkeeping method from one concern that must be composed (i.e. merged) with all or many methods from the other concern. A typical effect of n-ary redefinition is that it may result in code replication. In addition, the semantics of the crosscutting will likely demand additional definitions whenever the concern C_3 is extended. In these cases, this is again an example of composition anomaly.

4 Conclusion

In this paper we have presented the groundwork for further analysis and better understanding of the issues in developing and composing modular software from multiple concerns.

The first step to take in the development of software is to identify the concerns and take care of a clear separation of these concerns. For complex software, this will lead to the identification of multiple dimensions of concerns [Tarr 99].

In section 2, we have provided the requirements for composability. We denote components as *composable* if they are functionally and procedurally composable. Typically, explicit design efforts are required to achieve composable components.

Composable components, however, do not guarantee a successful composition of multiple components at the realization level. The composition scheme or the composition technique is a crucial factor in the realization of the composition. Depending on the characteristics of the composition scheme, a composition may fail, or the quality of the composition result, such as maintainability, may suffer (section 3 discussed this).

Finally, we have provided a definition for composition anomalies, which we repeat here informally:

> *Given two components that are both functionally and procedurally composable, if the realization of the composition for a given composition scheme is not possible or a composition is only possible with reduced quality aspects then we speak of a composition anomaly.*

Acknowledgements

This work has been partly funded by the 'Telematica Instituut', the Netherlands, in the scope of the AMIDST project, by the European Union, in the scope of the EASYCOMP project, and by the Dutch Organization for Sciences (NWO), in the scope of the SION programme. Our thanks go also to KPN Research in the Netherlands, who supported part of this work and provided a challenging problem for reasoning about composability issues.

References

[Matsuoka 90] S. Matsuoka, K. Wakita & A. Yonezawa, *Synchronization Constraints with Inheritance: What is Not Possible- So What is?*, Tokyo University, Internal Report, 1990

[Matsuoka 93] S. Matsuoka & A. Yonezawa, *Inheritance Anomaly in Object-Oriented Concurrent Programming Languages*, in Research Directions in Concurrent Object-Oriented Programming, (eds.) G. Agha, P. Wegner & A. Yonezawa, MIT Press, April 1993, pp. 107-150

[Tarr 99] P. Tarr, H. Ossher, W. Harrison & S.M. Sutton, Jr. *N Degrees of Separation: Multi-Dimensional Separation of Concerns.* In proceedings of ICSE 21, May, 1999.

[Vallecilo 99] A. Vallecilo, J. Hernandez & J.M. Troya. Object interoperability. In Object-Oriented Technology: ECOOP '99 Workshop Reader, number 1743 in LNCS, pages 1-21, Springer Verlag.