Research Topics in Composability

Carine Lucas, Patrick Steyaert, Kim Mens
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussels, BELGIUM
http://progwww.vub.ac.be/

Abstract

While composability is a much desired quality for software artefacts, there is no consensus whatsoever on what composability really is, nor on how it can be achieved. In this position paper we discuss three inhibitors of composability and hint at possible solutions. It is our conjecture that tackling these three problems is crucial to achieve true composability.

1. Inhibitor 1: Lack of a consistent terminology and insight in the relations between different composition techniques

Composability is not the only research area that focuses on building large and complex systems while emphasising modularity and separation of concerns. During the last few years, research in object-oriented software engineering has resulted in a proliferation of new terms and techniques that all strive for more composable, reusable, adaptable, … software:

- components often need to be *adapted* before they can be composed into some application;
- components can in some cases only be composed in some *framework*;
- components need an *open implementation* for tailorability;
- components can be stored as *reusable assets* in a reuse library;
- to improve compositionality sometimes part of the meta-system in which the components are expressed needs to be reified, hence the need for *computational reflection* and *meta-level architectures*;
- and so on, …

This abundance of new terms and techniques raises a series of questions. First, what does composability mean? When browsing through articles with composition in their titles, subjects as diverse as framework instantiation, CORBA IDLs, inheritance, … pop up. But the problem is not finding a clear definition for composition. What is important is to realise that issues like composability, adaptability and maintainability are all just subgoals. The main goal is to build good quality software that can be integrated with other systems and has the ability to evolve over time. Therefore to achieve more insight an effort should be made to develop a **classification of**

techniques, so that it becomes clear what different approaches do and do not offer, which problems they handle and how they overlap.

Even more important, to fully benefit from these new technologies we should be able to use them in combination. Therefore, a study of the **interactions between the different techniques** is called for. The interaction between reflection (and meta-level architectures) and increased compositionality is already actively being researched. For example, [Aksit&al.93] studies how object interactions can be abstracted in order to achieve a higher degree of reusability. [Demeyer96] studies how the combination of framework technology and meta-object protocols can be used to achieve different levels of tailorability (domain level, system level and configuration level) in open hypermedia systems. Combining framework technology and meta-object protocols shows that explicit representations of framework contracts should be part of a meta-object protocol, an insight that is helpful for the design of meta-object protocols.

While these are first steps towards an integration of different composition techniques, more efforts in this research area are necessary.

2. Inhibitor 2: Absence of composition interfaces that allow both flexibility and predictability of composition

To be able to compose components **clear composition interfaces** are needed. While this may seem evident, current practice shows that we are still far from having clear and unambiguously defined composition interfaces that allow the composition of components with a predictable outcome. On the one hand a strong emphasis has been put on *black box* composition. A black box interface only shows what functionality is provided, not how this functionality is achieved. Black boxes can be composed by plugging them together via their interface. While black box composition offers a guarantee that changes in the black box will not cause problems where it is used, black boxes often do not provide enough information to realise more sophisticated forms of composition.

The other extreme is *white box* composition, where you can see the inside and outside of the component and change it, e.g., through inheritance or delegation. While white box composition offers more possibilities for different types of composability, it also causes a bigger risk for users to be negatively affected by a change in one of the components. Furthermore, white boxes leave it to the user of the component to inspect the implementation of a component to see how it can be used. This is often a difficult job and therefore not without the risk of causing errors.

What we need is an intermediate solution. A kind of *grey box* interfaces, that allow independent programmers to use components without having to inspect all implementation details, but on the other hand do not hide so much information that usability gets too severely restricted. Therefore, these interfaces need to **provide all necessary information for the desired type of composition**, while **hiding**

unimportant implementation details. Grey box interfaces thus also allow component builders to restrict the ways in which a component can be used. It is clear that different types of composability call for different types of interfaces. For example, for using components through cloning, black box interfaces may suffice. For more complex composition mechanisms like inheritance or part-whole relationships other information is called for.

Let us investigate the example of inheritance in more detail. Inheritance is one of the oldest and most well-known composition techniques in OO. Still the development of composition interfaces for inheritance has started only recently. Inheritance is mainly seen as a mechanism to compose the behaviour of objects. It does so by composing independently developed packages of behaviour and combining them with an *inheritance mechanism*. A common characteristic of different inheritance mechanisms is *method overriding*. It allows for an inheritor component to adapt the behaviour of the parent component. While method overriding is fundamental to inheritance, it is also the source of many problems with respect to composition. The composition of a parent and child component can have unpredictable results as, for example, methods that are overridden in the inheritor methods can be accidentally invoked by methods in the parent component. (An example of another kind of problem will be given in section 3.) In an attempt to tackle these problems specialisation interfaces were introduced [Lamping93] (see also [Stata&Guttag95]). A specialisation interface describes which methods invoke which other methods in a class through self sends. This documentation can be used by programmers of subclasses to assess the effect of overriding a method, without having to inspect all implementation details of the parent classes.

As an example, consider an abstract class `Set` which defines a method `add` to add a single element to the set and a method `addAll` to add a collection of elements to the set simultaneously.

```
Class Set
   method add(Element) = 0
   method addAll(aSet:Set) =
     begin
      for e in aSet do
        self.add(e)
     end
end
```

When creating a subclass `CountableSet` of `Set` that keeps a count of the number of elements in the set, we need information on which methods depend on what other methods, in order to decide which methods need to be overridden. For example, if we know that `addAll` depends on `add` in its implementation, it is sufficient to override the method `add` to take counting into account. In [Steyaert&al.96] we introduce *reuse contracts* for classes to document exactly these dependencies. In such a reuse contract each method has a specialisation clause (in italics in the example below) that documents how it depends on the other methods from this reuse contract. The reuse contract is an interface description to which the implementation must comply.

```
Reuse Contract Set
   abstract
      add(Element)     {}
   concrete
      addAll(Set)      {add}
end
```

Reuse contracts for classes cannot only be used by inheritors to decide which methods need to be overridden but can also be used by parent classes to restrict the possible ways in which a component can be changed. For example, the reuse contract of the class `Set` above restricts possible subclasses by stating that the method `addAll` should always call (at least) the method `add`.

More complex components will need more complex interfaces. Early results exist on developing reuse contracts for interclass interactions (in analogy to *contracts* as defined by [Helm&al.90]), which show that the concept can be generalised to other structures than class hierarchies. Also, reuse contracts for state transition diagrams are currently under development. In general, reuse contracts are interface descriptions that document the protocol between developers and users of reusable components. The first aim of reuse contracts is to provide the necessary information on how components can be composed. They can also be used to provide different views on components for different users or to enlighten different aspects.

Reuse contracts can only be composed or adapted by means of certain predefined *reuse operators*. This gives the opportunity to specify the composition semantics on a more fine-grained level. For example, instead of composing classes merely by means of inheritance, reuse contracts for classes can be manipulated by means of reuse operators: concretisation, extension, refinement (and the inverse operators). Concretisation makes abstract methods concrete, extension adds new methods to a reuse contract and refinement refines the design of some methods by adding extra information to their specialisation clause. These reuse operators not only allow documenting the changes — and the intentions of these changes — made to a class, but a careful investigation of their interactions also allows to predict and manage the effect of these changes. This brings us to our third and last inhibitor.

3. Inhibitor 3: Lack of clear composition semantics that describe the intention of the composition and allow to manage change propagation

A final problem is that current **composition mechanisms** are too technical and too coarse-grained. It is our conjecture that it is not sufficient to have composition interfaces that describe the interface of a component. It is just as crucial to have information on the intention with which a component is composed with other components: does it extend or refine the behaviour of these other components, does it make other components more concrete, etc.? Below, we demonstrate through an example[1] that this information can allow the **detection of possible unpredictable or incorrect behaviour of the composed entity** by checking the compatibility of composed components. Moreover, the introduction of explicit reuse operators enables us to **manage the evolution of components**. One of the main benefits of composable software could exactly be that software that makes use of components can benefit from later improvements to the composed components. Managing the propagation of changes made to components so that (re-)users of that component are not invalidated remains one of the most essential problems in the development of composable software.

Let us focus again on our previous example of abstract classes as components and inheritance as composition technique. Suppose we want to make an optimised version `OptimisedSet` of `Set`. In this version `addAll` stores the added elements directly rather than invoking the `add` method to do this. This leads to inconsistent behaviour in `CountableSet` when `Set` is upgraded to `OptimisedSet`; additions made by `addAll` will not be counted anymore. This is because the assumption that `addAll` invokes `add`, whereon `CountableSet` implicitly depends, is broken in `OptimisedSet`. Using the terminology of [Kiczales&Lamping92] we say that `addAll` and `add` have become *inconsistent methods*. Although in this simple example the conflict can easily be derived from the code, in larger examples this is not so trivial. In practice it should be possible to detect such conflicts without code inspection. Reuse contracts and their operators provide the necessary information by making the assumptions made by adaptors explicit. For example, the reuse contracts of `CountableSet` and `OptimisedSet` document how they were derived from `Set`, and thus what assumptions about `Set` they rely on.

```
Reuse Contract CountableSet concretises Set
   concrete
      add(Element)
end

Reuse Contract OptimisedSet coarsens Set
   concrete
      addAll(Element) {- add}
end
```

The fact that `add` and `addAll` have become inconsistent can be detected directly by inspecting the reuse contracts. `OptimisedSet` is a coarsening (the inverse of a refinement) of `Set`, which means that it partially breaches `Set`'s design. This is done by removing a method from its specialisation clause (in italics above). `CountableSet` is a concretisation of `Set`, as it makes one of its abstract methods concrete. In general, inconsistent methods appear when a concretisation is performed of a method that has been removed from the specialisation clause of the exchanged parent by a coarsening.

---

[1] For a more detailed discussion and validation of our approach described here, we refer to [Steyaert&al.96]

This example illustrates that by examining the interactions between the different reuse operators and by investigating which operators respect the design, it becomes possible to detect attempts to compose incompatible components and to signal possible problems as, for example, inconsistent methods. Furthermore, rules can be constructed that assist the users of components in understanding how components can be used and in assessing the consequences of changes they make for systems that have already incorporated these components.

4. Conclusion

In this position paper we have discussed three inhibitors for true composability:

1. Lack of a consistent terminology and insight in the relations between different composition techniques.
2. Absence of composition interfaces that allow both flexibility and predictability of composition.
3. Lack of clear composition semantics that describe the intention of the composition and allow to manage change propagation.

We propose a solution based on the concept of reuse contracts and reuse operators:

In [Steyaert&al.96], reuse contracts have been developed for abstract classes as components and inheritance as composition technique. Currently reuse contracts are being developed for interclass interaction diagrams and state transition diagrams, which makes us believe that the same approach is applicable to other and more general composition techniques.

Reuse contracts are flexible interface descriptions that document the protocol between developers and users of reusable components. They provide detailed information on how components can be composed. Reuse contracts can be manipulated by means of reuse operators. As explained, reuse operators not only document the intentions of changes made to some component, but a careful investigation of their interactions also allows to predict and manage the effect of these changes.

References

[Aksit&al.93] Aksit, M, Wakita, K, Bosch, J, Bergmans, L and Yonezawa, A: *Abstracting Object Interactions Using Composition Filters*, In Rachid Guerraoui, Oscar Nierstrasz, and Michel Riveill, editors, ECOOP '93, Workshop on Object-Based Distributed Programming, number 791 in Lecture Notes in Computer Science, pages 152-184, Berlin Heidelberg, July 1993, ECOOP.

[Demeyer96] Demeyer, S: *ZYPHER: Tailorability as a Link from Object-Oriented Software Engineering to Open Hypermedia*, PhD thesis, Vrije Universiteit Brussel, July 96

[Helm&al.90] Helm, R, Holland, I M and Gangopadhyay, D: *Contracts: Specifying Behavioral Compositions in Object-Oriented Systems*, In Proceedings of ACM Joint OOPSLA/ECOOP'90 Conference Proceedings, pp.169-180, ACM Press 1990.

[Kiczales&Lamping92] Kiczales, G and Lamping, J: *Issues in the Design and Specification of Class Libraries*, Proceedings of OOPSLA '92, Conference on Object-Oriented Programming, Systems, Languages and Applications, pp. 435-451, ACM Press 1992.

[Lamping93] Lamping, J: *Typing the Specialization Interface*, In Proceedings of OOPSLA '93 Conference on Object Oriented Programming, Systems, Languages and Applications, pp. 201-214, ACM Press 1993.

[Stata&Guttag95] Stata, R and Guttag, J: *Modular Reasoning in the Presence of Subclassing*, In Proceedings of OOPSLA '92, Conference on Object-Oriented Programming, Systems, Languages and Applications, pp. 200-214, ACM Press 1995.

[Steyaert&al.96] Steyaert, P, Lucas, C, Mens, K and D'Hondt, T: *Reuse Contracts: Managing the Evolution of Reusable Assets,* To appear in Proceedings of OOPSLA '96 Conference on Object Oriented Programming, Systems, Languages and Applications, ACM Press 1996.