Verifying Correctness of Pattern-based Composition in Coq

Qiang Liu[1,1], Zongyuan Yang[1] and Jinkui Xie[1]

[1]Department of Computer Science and Technology, East China Normal University
Dongchuan Rd. 500, 200241 Shanghai, China
{lqiang, yzyuan, jkxie}@cs.ecnu.edu.cn

Abstract. Design patterns capture elegant design solutions and facilitate reuse in design level. In the Component Based Software Engineering (CBSE), design patterns were treated as design components, which serve as elemental components and can be composed to construct a large software system. In the process of composition, the key problem is how to ensure the correctness of composition. To address this problem, a criterion for correct composition is needed. The well known "faithfulness" principle is chosen as the correctness criterion and our discussions are based on this principle. In this paper, we at first use First Order Logic to model some elemental entities and relations in Object Oriented Design, which serve as an ontology in the domain. Then using the vocabulary in the ontology, we formally specify design patterns and formalize the "faithful" principle as theorems in Coq. Finally, we prove the theorems and thus show the correctness of composition. As a case study, we described and verified the composition of Composite pattern and Decorator pattern. Once a composition is proven to be correct, one can use the composition repeatedly. This would facilitate reuse of design in a larger scale and reduce errors in design phase, which justifies all the efforts of verifying their correctness.

Keywords: Design patterns, Composition, Verification, Coq, Faithfulness

1 Introduction

Design patterns [8] are widely used in modern software systems. In the Component-Based approach, design patterns were treated as design components, which serve as elemental components and can be composed to construct a large software system. In the process of composition, the key problem is how to ensure the correctness of composition. This is a difficult and larger top, and far from being resolved. In this paper, we set our discussion to the domain of the object oriented design, and restrict our discussion in the composition of design patterns. This paper is organized in the following way:
Section 2 is related works.

In section 3, we illustrate the problem by a case study, the composition of Composite and Decorator pattern. In order to model design patterns in Coq, we use First Order Logic to capture the basic entities (Class and Method) and relations in the domain of OOD (Inherit etc). They are both modeled by Inductive types in Coq. These entities and relations are the basic vocabularies based on which we can faithfully

describe design patterns. The novel thing in our way is that we model design patterns as a list of "propositions", instead of the conjunction of propositions. This can greatly simplify our definitions in latter works. Once we can model design patterns in lists, we can easily describe their composition by concatenating lists. This is the bases for latter works.

In section 4, we give a formal proof of the composition of Composite and Decorator pattern. First, as the composition of design patterns are modeled as lists, the problem of verifying correctness of the composition can be tackled by verifying certain properties of these lists. In this way, we successfully formalized the "faithful" principle as two theorems about "lists" in Coq. Then we devise a couple of customized tactics to facilitate the proof of these theorems. Our tactics take lists (design patterns) as parameters, thus applicable to all the patterns modeled in the correct way (as lists). By using our tactics, we prove that the composition in our example satisfies the properties required by these theorems, which means it is a correct composition in the sense that it is a faithful composition.

Section 5 is conclusion and future works.

To sum up, we have developed a new way of modeling design pattern (as a list of proposition). Based on this model, we can describe their composition as concatenation of lists and formalize properties of compositions as properties of lists, which are easier to operate and verify. In the case study of Composite and Decorator pattern composition, we have shown that the correctness of pattern based composition can be verified with the assist of Coq. Also, by using our carefully designed tactics, the proving processes can be done with a high level of automation.

All the materials presented in this paper are implemented and compiled in Coq v8.1 and available at [14]. Coq [12] is an interactive theorem proving and program development tool, which is widely used in program verification and theorem proving.

2 Related Works

The formalization of design patterns was widely studied [3][5][16], and the works that study their composition are also immense [1][2][4][6][7][15]. [1] studied composition in general, and [2] proposed the concept of 'faithfulness'. Their works can apply to any domain, but in this paper, we only focus on the domain of Object Oriented Design. In [5], the authors formulated basic entities and relations to model the basic concepts in OOD, which is similar to our work, but [5] mainly devoted to graphical representations of OOD to facilitate understanding of system design, and do not considered pattern based compositions. [4][6][15] studied pattern based composition, and used First Order Logic and Temporal Logic of Action to describe design patterns' structure and behavior respectively. Both of them chose the 'faithfulness' principle as correctness criteria, and analyzed the conditions that a correct composition must satisfy. But their proofs are informal and manual, which is based on observations and experiences. We adapt Coq as the proof assistance in this paper, which makes our proof formal and gained a high level of automation.

3 Problem Illustration

In Object Oriented Design, Class and Method are identified as basic entities, and a set of primary relations were formulated to model the basic concepts in OOD. The number and name of these relations might differ from one another [4][5][6]. In our approach, we define the following inductive type in Coq as in Figure 1. It is easy to add new relations by adding new clauses in the inductive definition.

```
Inductive term : Set :=
| Inherit (x :Class) (y : Class) : term                    (* Class x inherit
from Class y *)
| Memberfun (m:Method) (c:Class): term                     (* Method m is a
member function of Class c *)
| Reference (x:Class)(y:Class):term
| Invoke(c1:Class)(m1:Method)(c2:Class) (m2:Method) : term
                        (* Method in Class c1 invoke Method m2 in Class
                        c2 *)
| Argument (m:Method)(c:Class):term                  (*reference of Class c is
an argument of Method m *).
```

Fig. 1. Inductive definition of term

Words in (* *) are comments. This definition serves as a grammar of writing specifications about design patterns. Usually, it will take many terms to precisely describe a design patter, so we model it as a list of 'term':

Definition DesignPattern:= list term. (1)

The benefit of this definition is fully discussed in Section 1. Now, as a case study, consider the composition of Composite pattern and Decorator pattern. Their descriptions in UML class diagram are presented in Figure 2 and 3.

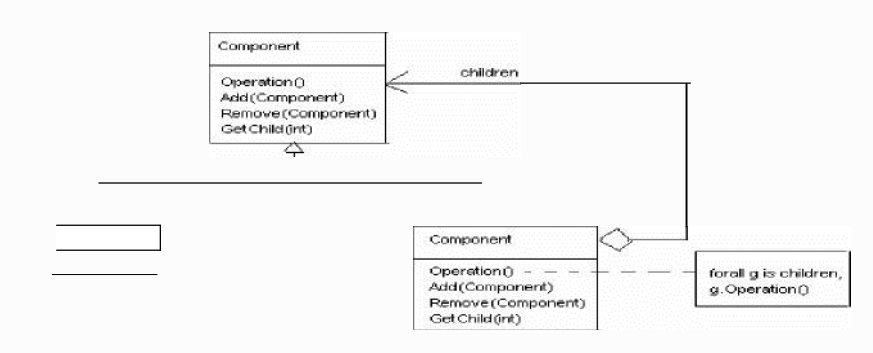

Fig. 2. Composite pattern in UML

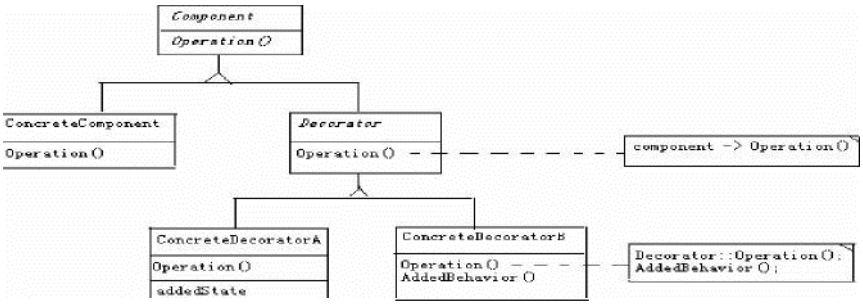

Fig. 3. Decorator pattern in UML

By using these definitions, we can formally define Composite pattern and Decorator pattern by using a list of 'propositions' as in Figure 4 and Figure 5.

```
Definition composite_pattern : DesignPattern :=
  Memberfun operation component
::Memberfun add component::Memberfun remove component
::Memberfun getchild component
::Memberfun operation composite::Memberfun add composite
::Memberfun remove composite::Memberfun getchild composite
::Reference composite component
::Invocate composite operation component operation
::Memberfun operation leaf::Argument add component
::Argument remove component::Argument getchild int
::Inherit leaf component::Inherit composite component::nil.
```

Fig. 4. Definition of Composite pattern in Coq

In practice, when software designers want to compose these two patterns, depending on their experiences and intuitions, they may compose them as in Figure 6 (in UML class diagram). In fact, we can treat this composition as a new pattern; we call it the Composite_Decorator pattern. Its formal counterpart is in Figure 7.

However, we cannot rely on the experiences and intuitions of the designer, because this work is laborious and error-prone. First, there are many statements in each pattern, the designer may lose statements in the original patterns, or add undesirable new statements in the composition. Additionally, as we can see in Figure 6, there is a mapping between entities in the separate design patterns and entities in the composition, because some entities play roles in both separate patterns. Therefore, we need a tool to maintain all these information for the designer and prove the correctness of their compositions. That is the main job in the next section.

```
Definition decorator_pattern : DesignPattern :=
    Memberfun operation component::Memberfun operation concretecomponent
::Memberfun operation decorator::Reference decorator component
::Invocate decorator operation component operation
::Memberfun operation concretedecoratora::Reference concretedecoratora
addedstate
::Memberfun operation concretedecoratorb::Memberfun addedbehavior
concretedecoratorb
::Invocate concretedecoratorb operation decorator operation
::Invocate concretedecoratorb operation concretedecoratorb addedbehavior
::Inherit concretecomponent component::Inherit decorator component
::Inherit concretedecoratora decorator::Inherit concretedecoratorb decorator::nil.
```

Fig. 5. Definition of Decorator pattern in Coq.

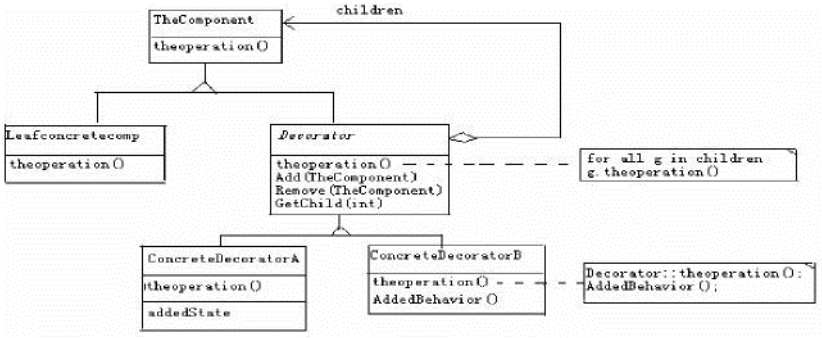

Fig. 6. The composition of Composite and Decorator in UML.

4 The Formal Proof

To prove the correctness of composition, we first need a correctness criterion. We choose the well known principle of 'faithfulness' as our correctness criterion. In general, for a composition to be faithful, it must satisfy the follow two conditions:

No component loses any properties after composition.

No new properties about each component can be added after composition.

As to our case study, this means: for the composition of Composite and Decorator pattern to be correct, it must satisfy the following two conditions:

No statement in Composite and Decorator loses after the composition.

No new properties about Composite and Decorator can be added after the composition. Now we explicitly define mappings between entities, before and after the composition. To prove the composition is faithful under these mappings, a set of mapping functions are defined in Figure 8.

```
Definition composite_decorator_composition: DesignPattern := Memberfun
theoperation thecomponent::Memberfun add thecomponent ::Memberfun
remove thecomponent::Memberfun getchild thecomponent ::Memberfun
theoperation leafconcretecomp::Memberfun theoperation
compositedecorator::Memberfun add compositedecorator
::Memberfun remove compositedecorator::Memberfun getchild
compositedecorator::Reference compositedecorator thecomponent
::Memberfun theoperation concretedecoratora::Reference concretedecoratora
addedstate::Memberfun theoperation concretedecoratorb::Memberfun
addedbehavior concretedecoratorb:: Invocate concretedecoratorb theoperation
decorator operation::Invocate concretedecoratorb theoperation
concretedecoratorb addedbehavior::Argument add thecomponent::Argument
remove thecomponent::Argument getchild int::
Inherit leafconcretecomp thecomponent::Inherit compositedecorator
thecomponent::Inherit concretedecoratora compositedecorator::Inherit
concretedecoratorb compositedecorator::nil.
```

Fig. 7. Definition of composition of Composite and Decorator in Coq

```
CM : Class -> Class;
MM :Method ->Method; Mcmp:Composite->Composite_Decorator ;
Mdcr:Decorator->Composite_Decorator ; Mcmp_dcr := Mcmp \/ Mdcr.
```

Fig. 8. Mapping functions

Now we formalize the principle of faithfulness in Coq. As mentioned in Section 1, the theorems are actually properties about lists, which is easier to state and verify in Coq.

$$\forall(\text{t:term}),\ \text{In t (Composite} \wedge \text{Decorator)} \rightarrow \exists\,(\text{s:term}),\text{In s Composite\_Decorator}/\backslash(\text{Mcmp\_dcr t=s}). \quad (2)$$

$$\forall(\text{t:term}),\ \text{In t Composite\_Decorator} \rightarrow \exists\,(\text{s:term}),\ \text{In s (Composite} \vee \text{Decorator)}/\backslash\ (\text{Mcmp\_dcr s=t}) \quad (3)$$

(Composite$\vee$Decorator) is not only the disjoint union of Composite and Decorator, it is the deductive closure of their union. The closure is calculated by our function "Union" and "Closure" in Coq. For details and comments, see [14]. Usually, we allow some new facts to be deduced from the union of two patterns, with explicitly specified rules. Otherwise, any other new facts would be undesirable, and would be regarded as an error in the composition. In our example, there is only one rule allowed for reasoning, that is:

$$\text{Inherit a b, Inherit b c=> Inherit a c.} \quad (4)$$

Other rules can easily be added to our definition ('fsttrans') by simply adding a new clause in its definition. To facilitate proof of these theorems, we defined customized tactics. These tactics allow some automation in the proving process. See

the proof in Figure 9。

```
Theorem not_lose_Mcmp_dcr : forall(t:term), Int composite_decorator->
(existss:term, In s composite_decorator'/\Mcmp_dcr t=s).
Proof. not_lose_tac composite_decorator composite_decorator' Mcmp_dcr. Qed.
Theorem not_add_Mcmp_dcr : forall (t:term), In t composite_decorator' ->(exists
s:term, In s composite_decorator/\Mcmp_dcr s=t).
Proof. not_add_tac composite_decorator' composite_decorator. exists (Inherit
concretedecoratora component). auto 40 .
exists (Inherit concretedecoratorb component ). auto 40. Qed.
```

Fig. 9. The proof of not lose theorem

Till now, we have demonstrated our formal way step by step, and have proven that the example composition is a correct one. By following our way, designers can prove the correctness of their own composition. Once a composition is proven to be correct, it can be reused repeatedly in the process of system construction. This would facilitate reuse of design in a larger scale and reduce errors in design phase, which justifies all the efforts of verifying their correctness.

5 Conclusions and Future Work

The main contribution of this paper is to adopt Coq as a tool to formally verifying the correctness of pattern-base composition and provide some basic functions to facilitate the proving process. By using our basic definitions, designers can conveniently formulate desirable properties of design patterns as theorems in Coq and then prove their correctness. This will facilitate reuse in a larger scale (not only reuse of design pattern, but also their composition). The reuse of compositions can benefit many aspects in the developing process, including improving developing efficiency (using larger building blocks) and reduce errors in design phase (the composition is proven to be correct). Additionally, the composition is a larger logic unit (like the Composite_Decorator pattern), which makes the system easier to understand and maintain.

However, though for some patterns it is enough to focus on their structural aspects, some design patterns do have behavioral significance, like Observer pattern. The relations in our definition of ‘term’ in Coq are all permanent relations (once coded in the program, they cannot be changed during run time). To model behavioral aspects of design patterns, one will have to use temporal logic, like Temporal Logic of Action [9]. It is our future concern to integrate these two aspects in Coq. Furthermore, to achieve a higher level of automation in the proving of composition, we should develop a GUI tool to convert UML diagrams to Coq scripts. This would hide details about Coq from designers, thus improve the work efficiency of system designers.

Aknowledgements. The research in this paper was supported by Research Fund for the Doctoral Program of Higher Education of China, No. 20060269002.

References

1. Abadi, Martin, Lamport, Leslie.: Composing specifications. ACM Transactions on Programming Languages and Systems 15 (1),73–132 (1993)
2. Moriconi, Mark, Qian, Xiaolei, Riemenschneider, R.A.: Correct architecture refinement. IEEE,Transactions on Software Engineering 21 (4),356–372 (1995)
3. Mikkonen Tommi.: Formalizing design pattern. In: Proceedings of the 20th International Conference on Software Engineering, 115–124 (1998)
4. Taibi, Toufik, Ngo, David C.L.: Formal specification of design pattern combination using BPSL. International Journal of Information and Software Technology (IST) 45 (3), 157–170 (2003)
5. Eden, A.H., Hirshfeld, Y.: Principles in formal specification of object-oriented architectures. In: Proceedings of the 11th CASCON, November 2001, Toronto, Canada (2001)
6. Dong, Jing, Alencar, Paulo, Cowan, Donald.: Ensuring structure and behavior correctness in design composition. In: Proceedings of the 7th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS), Edinburgh UK, 279–287 (2000)
7. Keller, Rudolf K, Schauer Reinhard.: Design components: towards software composition at the design level. In: Proceedings of the 20th International Conference on Software Engineering, 302–311 (1998)
8. Gamma, Erich, Helm, Richard, Johnson, Ralph, Vlissides, John.: Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley Publishing Company (1995)
9. Lamport, Leslie.: The temporal logic of actions. ACM Transactions on Programming Languages and Systems 16 (3), 873–923 (1994)
10. Thierry Coquand, Gerard Huet.: The calculus of constructions.Technical Report 530, INRIA, 1986.
11. Jean-Yves Girard, Yves Lafont, Paul Taylor.: Proofs and Types, volume 7 of Cambridge Tracts in Theoretical Computer Science.Cambridge University Press (1989)
12. Yves Bertot, Pierre Cast´eran.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. Springer Verlag (2004)
13. P. Alencar, D. Cowan, C. Lucena.: A formal approach to architectural design patterns. In Proceedings of the 3rd International Symposium of formal Methods Europe, 576 – 594 (1996)
14. Pattern Composition in Coq, http://ecnucs-selab.googlecode.com/files/exampleinpaper
15.Jing Dong, Paulo S.C. Alencar, Donald D. Cowan , Sheng Yang.: Composingpattern-based components and verifying correctness. The Journal of Systems and Software ,1755–1769 (2007).
16. Amnon H. Eden, Yoram Hirshfeld, Rick Kazman.: Abstraction classes in software design. IEE Software, Vol. 153, No. 4, 163–182 (2006)