The Impact of Pattern Use on Design Quality

Position Paper for the OOPSLA 2001 Workshop "Beyond Design: Patterns (mis)used"

Ralf Reißing
Institute of Computer Science, University of Stuttgart
Breitwiesenstr. 20-22
70565 Stuttgart, Germany
reissing@informatik.uni-stuttgart.de

There is in all things a pattern that is part of our universe. It has symmetry, elegance, and grace - those qualities you find always in that which the true artist captures. You can find it in the turning of the seasons, in the way the sand trails along a ridge, in the branch clusters of the creosote bush or the pattern of its leaves. We try to copy these patterns in our lives and our society, seeking the rhythms, the dances, the forms that comfort. Yet, it is possible to see peril in the finding of ultimate perfection. It is clear that the ultimate pattern contains its own fixity. In such perfection, all things move toward death. (Frank Herbert, Dune)

Abstract

The pattern community promises that using design patterns enhances design quality. So, if we have two similar designs A and B for the same problem, B using design patterns and A not using design patterns, B should have higher quality than A. However, if we apply "classic" object-oriented design metrics to both designs, the metrics tell us that design A is better – mostly because it has less classes, operations, inheritance, associations, etc.

So the question is: who is wrong? The metrics or the pattern community? Do we have the wrong quality metrics for object-oriented design? Or does using patterns in fact make a design worse, not better?

1 The Problem

The correct use of design patterns is supposed to increase the quality of designs. Because design quality can be measured by quality metrics, the use of design patterns should lead to better measures. However, many common object-oriented design metrics indicate lower quality if design patterns are used. The following example shows such a case. Consider the following two design alternatives, one of them using patterns. The example was inspired by a case study in Martin Fowler's book on refactoring ([2], chapter 1).

1.1 Scenario

A video store that rents movies to customers needs a software system for printing the statement of customer's charges. The requirements have been clarified, now the system has to be designed in an object-oriented way.

The concepts of customer, rental, and movie are modeled as classes. A customer has rentals, while each rental knows the rented movie and the number of days it was rented. Each movie has a price code; the charge for a rental is determined by the price code of the movie and how long it was rented. The price codes are: regular (normal price), children's (children's movies are cheaper), and new release (newly released movies are more expensive). The price code of a movie can change, e.g. from new release to regular. Therefore the class Movie has a method called setPriceCode.

1.2 Design A

Figure 1 shows the class diagram of design A and the sequence diagram for the statement method that prints a statement. The charge for each rental is calculated by the Movie object that knows its price code and is forwarded the number of days rented from Rental. The price codes are modeled by an enumeration type.

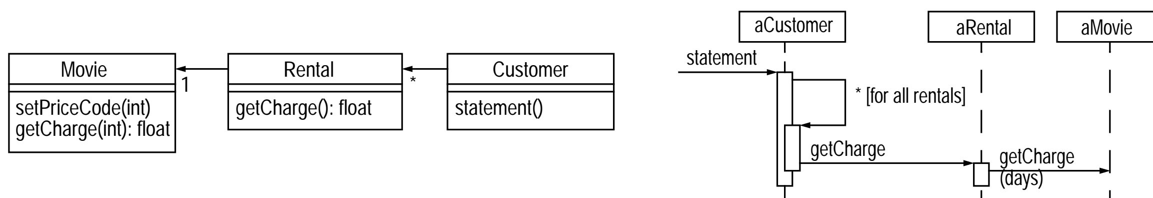

Figure 1: Class and sequence diagram for design A

1.3 Design B

Design B introduces the State pattern [3] into design A. The enumeration type for the price code is replaced by the abstract class Price and its subclasses. For each price code there is a subclass of Price that calculates the charge when given the number of days the movie was rented. The resulting design and the sequence diagram for the statement method are shown in figure 2.

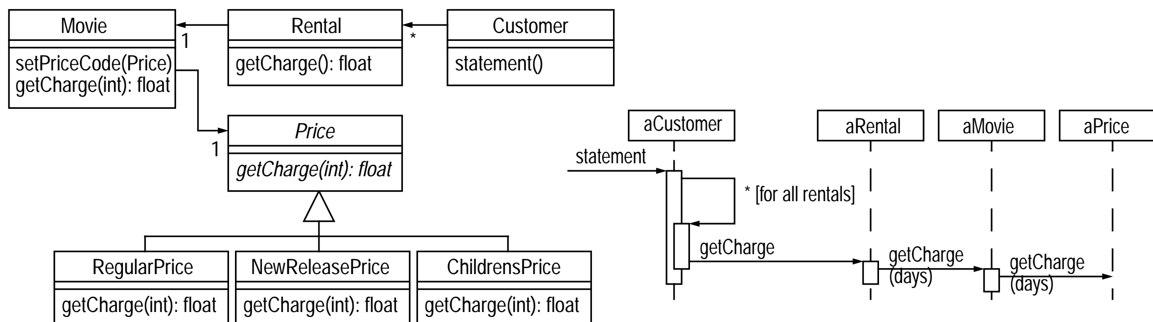

Figure 2: Class and sequence diagram for design B

Now it is easier to add new price codes, because only a new Price subclass has to be added – in contrast to design A, where the enumeration type and the getCharge method of Movie had to be modified. So the design quality, especially the flexibility, has improved.

1.4 Metrics

Now we apply some design metrics to the two design alternatives. First, some common size metrics, then the famous Chidamber and Kemerer metrics suite [1].

1.4.1 Size Metrics

Size metrics can be applied to classes (e.g. number of attributes, methods, and relationships) and to the whole design (e.g. number of classes). Table 1 shows some class size metrics. Increasing size is supposed to increase complexity, therefore to decrease design quality. Therefore higher measures indicate lower quality. In order to be able to count the attributes, some assumptions were made: Movie has a private attribute to store the price code, Rental has an attribute to store the duration of the rental.

The metrics show an improvement in NOA of Movie, but because the attribute is replaced by an association, NOD gets worse. Overall this can be considered a change for the worse because an additional dependency causes higher coupling which has more influence on quality than size itself. The size of the design as a whole gets far worse: The number of classes is more than doubled from three classes in design A to seven in design B.

1.4.2 Chidamber and Kemerer Metrics

In order to calculate these design metrics, detailed design information is needed: which method accesses which attributes and calls which methods. The method calls can be derived from the sequence diagram, but not the accesses to attributes. Therefore assumptions are made which methods access which attributes, including the additional attributes needed for implementing the associations. Table 2 shows the resulting measures. Chidamber and Kemerer consider their metrics to be complexity metrics, so higher values indicate lower quality.

	NOA[a]		NOM[b]		NOD[c]	
	A	B	A	B	A	B
Movie	1	0	2	2	0	1
Rental	1	1	1	1	1	1
Customer	0	0	1	1	1	1
Price	-	0	-	1	-	0
RegularPrice	-	0	-	1	-	1
NewReleasePrice	-	0	-	1	-	1
ChildrenPrice	-	0	-	1	-	1

Table 1: Class size metrics for designs A and B

a. NOA (number of attributes): number of locally defined attributes
b. NOM (number of methods): number of locally defined methods
c. NOD (number of dependencies): number of relationships that cause a dependency (e.g. association, inheritance)

DIT and NOC show that little use is made of inheritance, which is typical for such small examples. The measures of the classes present in both designs change very little, but if they change, they change to the worse (CBO and RFC of Movie).

	WMC[a]		DIT[b]		NOC[c]		CBO[d]		RFC[e]		LCOM[f]	
	A	B	A	B	A	B	A	B	A	B	A	B
Movie	2	2	0	0	0	0	0	1	2	3	0	0
Rental	1	1	0	0	0	0	2	2	2	2	0	0
Customer	1	1	0	0	0	0	1	1	2	2	0	0
Price	-	1	-	0	-	3	-	1	-	1	-	0
RegularPrice	-	1	-	1	-	0	-	0	-	1	-	0
NewReleasePrice	-	1	-	1	-	0	-	0	-	1	-	0
ChildrenPrice	-	1	-	1	-	0	-	0	-	1	-	0

Table 2: Chidamber and Kemerer metrics for designs A and B

a. WMC (weighted methods per class): here: number of methods, i.e. weight factor is 1
b. DIT (depth of inheritance tree): depth of the class in the inheritance hierarchy
c. NOC (number of children): number of direct subclasses
d. CBO (coupling between object classes): number of classes the class is coupled to (by uses-relationships)
e. RFC (response for a class): number of methods of the class plus number of methods of other classes called
f. LCOM (lack of cohesion in methods): measure for the lack of cohesion of methods and attributes in a class

1.4.3 Observation

According to the design metrics used design B has lower quality than design A – despite the fact that the use of the State pattern in design B is appropriate and enhances maintainability. Obviously there is a price to pay for the additional flexibility – and it shows in the traditional design metrics, whereas the improvement does not show that much. This effect was also been discussed at the metrics workshop at ECOOP 2001, where many participants stated they had observed it.

2 Solution

2.1 Analysis of the Problem

So what is the cause of the contradiction between supposed the quality improvement by design patterns and the measured quality deterioration?

First of all, obviously there is a definition problem for the term “quality”. If we define quality to consist solely of flexibility, design B is indeed better than A. If we consider quality to be only size and coupling, design A is the better one. So what we need is a more comprehensive definition of quality.

Second, it is a question of what is measured. If we want to favor design patterns, we could use a “number of design pattern uses” metric to indicate quality. The more design pattern uses are in a design, the better it is. Unfortunately, things are not that simple. Design patterns can also be abused if their context and forces are not properly considered – a common mistake by beginners who think that the use of design patterns in itself is good. Therefore this metric alone is a bad quality indicator.

2.2 Suggestion for a Solution

A more appropriate notion of quality should result if the quality definition includes both views, the traditional design metric view based on size, coupling, and other complexity criteria and the flexibility considerations inherent in design patterns. For example, if the use of design pattern metric is balanced with other metrics, e.g. size and coupling metrics, it is much more useful.

Currently I am working on a quality model for object-oriented design (called QOOD, see [4]) that combines traditional quality measurement (the hard facts) with expert judgements on criteria that are difficult to measure (like flexibility). This “holistic” design quality assessment incorporates the benefits of both qualitative and quantitative quality aspects and should result in better quality assessments, especially when design patterns are involved.

Of course, it is the same with object-oriented design and design patterns as with any other technology: Before designers can use it properly and are able to produce high quality it takes a lot of training – and experience. The QOOD quality model should help to set the course towards high design quality.

References

[1] Chidamber, S.; Kemerer, C.: A Metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering, 20(6), 1994, 476-493.

[2] Fowler, M.; Beck, K.; Brant, J.; Opdyke, W.; Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading, MA, 1999.

[3] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, 1995.

[4] Reißing, R.: Assessing the Quality of Object-Oriented Designs. To be published in the OOPSLA 2001 proceedings.