Study Programmers Preferences for Object Extensions in Object-Oriented Programming

Polina Galishnikova
pkgalishnikova@edu.hse.ru
Higher School of Economics
Moscow, Russia

Abstract

In object-oriented programming, additional functionality may be added to classes with the help of inheritance, decoration, composition, or simply making existing classes larger by adding new code or new methods to them. It is expected that most of programmers, especially those with more than 10 years of practical coding experience, choose decoration or composition, since these approaches lead to better design. In order to validate this intuition, we suggest conducting a survey among a reasonably large group of programmers, showing them a number of code snippets and asking to chose a method for their modifications.

1 Introduction

The issue of code reusability can be defined as the main one in the field of computer software. As Myung-Hoon Chung mentioned in the article "Science Code .Net: Object-oriented programming for science" [2], even in scientific computing the problem is acute. Object-oriented programming is one of the key development paradigms. It provides a plenty of benefits to programmers, as the programs written according to the object-oriented programming principles are more structured, easier to understand and adjustable, despite the fact that sometimes they can be more complicated in creating. [6]

One of the main advantages of object-oriented programming is the ability to use objects and methods of already written classes. Extension methods help programmers add additional functionality to classes. By creating extended classes based on simple ones, programmers save time and efforts during development, as well as greatly simplify the search and correction of errors in the program. They may be useful in cases when there is no access to some fragments of code, they cannot be changed for some reason or it is needed to add functions without sub-classing. There are plenty of ways to extend objects in object-oriented programming, however not all the programmers know, which of the approaches will lead to a better design and make code more efficient and which of them may result in complications.

The subject is quite relevant, as object-oriented programming became a fundamental paradigm in software development and it is essential to sustain code patterns, widely accepted among programmers, in order to conveniently modify pieces of code without unnecessary sophistication. [3] [7]

It is expected that most of programmers, especially those with more than 10 years of practical coding experience, choose decoration or composition, since these approaches lead to better design. In order to validate this intuition, we want to conduct a survey regarding existing Object Extension approaches and programmers' preferences and analyze the results.

2 Method

The goal of this study is to understand programmers' preferences for object extensions. This leads to the following research questions:

RQ1: What approaches exist in Object Extensions?

RQ2: What is the correlation between the factors (i.e. programmer's experience, skills, favourite programming language) and the programmer's choice?

At first, we selected most-used object-oriented extension methods by studying guidebooks on object-oriented programming, researching existing samples of code and communicating with programmers. It was required in RQ1, as we have to understand which methods of Object Extensions endure.

Secondly, based on this data, we created a questionnaire with code snippets, in which additional functionality is added involving multiple methods.

Thirdly, we found 95 programmers with varying experience, who would like to participate in survey. Second and the third step combined provided us with a base to answer RQ2.

Fourthly, programmers were given a number of snippets of code and were asked to chose a method for their modifications. This was motivated by both RQ1 and RQ2; we had to understand if found approaches were in fact used in practice and which of them were more preferable. We assume that providing options for modifying code would represent the preferences better.

Finally, we collected the results obtained from the questionnaire. Analysis of the data showed correlations between RQ1 and RQ2 and provided results to the survey.

3 Results

Extension methods allow to add new methods or functionality to already existing data structures without modifying the code. It is especially useful, in case we have to add new method to already existing structure, but for some reason we

cannot change the data or we do not have access to it. There are several methods of Object Extensions in object-oriented programming.
Bertrand Mayer in his book "Object-oriented Software Construction" [5] describes several approaches of object extension. He mostly focuses on inheritance and static methods. What is more, in the book "Design Patterns: Elements of Reusable Object- Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides [4] designated such design patterns as Facade, Composite and Decorator, which can be used to extend objects. Finally, G. Antoniol describes delegation pattern in object-oriented programming in his article "Object-oriented design patterns recovery". [1]
We developed a class of geometric figures *Figure*, which had a parameter *side* and a function *getSide*, which returned parameter *side*. The goal of using object extensions was to add additional functionality to the class by adding a method, which calculated the area of a figure.

```
class Figure {
public:
    int side;
    Figure (int side) : side(side) {}
    virtual int getSide() {
        return side;
    }
};
```

Static methods: here, in the sample of the code, let us define derived class *Square*, in which we develop static function *area*, which return the area of the square.

```
class Figure {
public:
    int side;
    Figure (int side) : side(side) {}
    virtual int getSide() const {
        return side;
    }
};

class Square : public Figure {
public:
    Square(int side) : Figure(side) {}
    static int area(const
    Figure& figure) {
        return figure.getSide() *
        figure.getSide();
    }
};
```

Decorator: in this code sample we implemented a Decorator Pattern. *SquareDecorator* complements both of the previous classes and enhances the functionality of a *Figure* object. This decorator class encapsulates a *Figure* object, enabling the calculation of the square's area through the *area* method.

```
class Figure {
public:
    int side;
    Figure (int side) : side(side) {}
    virtual int getSide() {
        return side;
    }
};
class Square : public Figure {
public:
    Square(int side) : Figure(side) {}
    int getSide() override {
        return side;
    }
};
class SquareDecorator {
public:
    SquareDecorator(Figure* figure) :
    figure(figure) {}
    int area() {
        return figure->getSide() *
        figure->getSide();
    }
private:
    Figure* figure;
};
```

Composition: here in the code sample we define class Figure and inherited class Square. A *Composition* object has a *Square* object as part of its internal state, and the *Composition* object can use the *Square* object to perform operations related to the composition's dimensions.

```
class Figure {
public:
    int side;
    Figure (int side) : side(side) {}
    virtual int getSide() {
        return side;
    }
};
class Square : public Figure {
public:
    Square(int side) : Figure(side) {}
    int getSide() override {
        return side;
    }
};
class Composition {
```

```
public:
    Composition(int side) : square(side) {}
    int area() {
        return square.getSide()*
        square.getSide();
    }
private:
    Square square;
};
```

Inheritance: the *Square* class is derived from the parent class using inheritance. It has a member function *area* that calculates and returns the area of a square, which is achieved by calling the *getSide* function inherited from the *Figure* class

```
class Figure {
public:
    int side;
    Figure (int side) : side(side) {}
    int getSide() {
        return side;
    }
};

class Square : public Figure {
public:
    Square(int side) : Figure(side) {}
    int area() {
        return (getSide() * getSide());
    }
};
```

Delegation: the *area* method of the *Area* class calls the eponymous method of the *Square* object and returns the result. The code structure is assembled in a way to delegate functionality of *Square* class to the class *area*.

```
class Figure {
public:
    int side;
    Figure (int side) : side(side) {}
    virtual int getSide() {
        return side;
    }
};
class Square : public Figure {
public:
    Square(int side) : Figure(side) {}
    int area() {
        return (getSide() * getSide());
    }
};
class Area {
public:
    Square square;
    Area(Square square) : square(square) {}
    int area() {
        return square.area();
    }
};
```

Facade: the *Facade* class simplifies interactions with *Square* and *Rhombus* objects. It contains instances of *Square* and *Rhombus* and the constructor which initializes them. The *Facade* class encapsulates the complexity of calculating the areas, shielding clients from the implementation details of the *Square* and *Rhombus* classes and allowing them to just create an object of *Facade* class in main function and quickly get results.

```
class Figure {
public:
    int side;
    Figure (int side) : side(side) {}
    virtual int getSide() {
        return side;
    }
};
class Square : public Figure {
public:
    Square(int side) : Figure(side) {}
    int getSide() override {
        return side;
    }
};
class Rhombus : public Figure {
public:
    Rhombus(int side) : Figure(side) {}
    int getSide() override {
        return side;
    }
};
class Facade {
public:
    Square square;
    Rhombus rhombus;
    Facade (Square square, Rhombus rhombus)
    : square(square),
    rhombus(rhombus) {}
    double sinusAlpha = 0.5; //constant
    is needed to obtain area of rhombus
    int SquareArea() {
        return square.getSide() *
        square.getSide();
    }
    double RhombusArea() {
        return rhombus.getSide() *
        rhombus.getSide() *
        sinusAlpha;
    }
};
```

4 Discussion

Combining all of these code samples, a survey was created. Overall, the number of participants was 95. It is a bit less, than expected, however I believe that it was enough to test the initial hypotheses.

In the survey participated 6 programmers with programming experience less than 1 year. The highest share is from 1 to 3 years of experience, having 28 programmers. From 3 to 5 years of experience there were 17 participants, 20 participants with experience from 5 to 10 years and 24 programmers with more than 10 years experience.

At first I visualized the distribution of votes for and against a

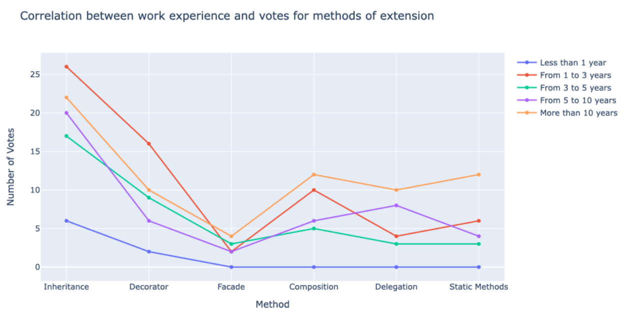

Figure 1. Correlation between work experience and votes for methods of extension

certain method, which turned out to be quite illustrative. The most popular object extension approaches are Inheritance and Decorator, having 91 and 43 votes for them respectively. However, there is a considerable gap between them - there are almost twice as many votes for inheritance. The least preferable options is facade - it has 84 votes against. The 'No' votes for Delegation and Static methods are evenly distributed - 70 for each of the method. These results suggest that developers generally favor Inheritance over other object extension methods, likely due to its simplicity and familiarity. In spite of the fact that it is still popular, Decorator lags behind significantly. The low preference for Facade may be caused by its complexity and potential for creating tightly coupled code.

Then I validated an initial hypothesis regarding the preferences of programmers with different levels of practical coding experience when it comes to object extension methods. The hypothesis states that most programmers, especially those with more than 10 years of experience, choose decoration or composition over inheritance, as these approaches lead to better design.

The initial hypothesis was partly true. The graph shows that programmers with more than 10 years of practical coding experience prefer inheritance, composition and static methods, with inheritance having an undisputed lead. Decorator, on the other hand, is not that widespread, having only 10 of the votes (compared to 22 for inheritance and 12 for composition). Programmers highlight that inheritance is more familiar, obvious and simple, allowing the code to be understood by larger number of developers.

The situation regarding programmers with 5 to 10 years of experience is the following: the most preferable method is, again, inheritance with 20 votes; however, the second place is taken by delegation, having 8 votes. It is noteworthy that almost all voters in this segment are currently employees at the Central Bank of Russia.

Developers with coding experience from 3 to 5 years mostly chose inheritance, decorator and, less frequently, composition. Façade, delegation and static methods were not in favor, having only 3 votes each.

The higher share of survey participants (group from 1 to 3 years) showed the highest contribution to inheritance and decorator. Composition is preferred as well. One voter in this segment provided an explicit commentary: "The inheritance method suits for extending class functionality when defining similar objects. For instance, circle, square, or rectangle can be considered as geometric figures. It is not possible to identify which figure exactly we are working with just by using the 'Figure' class. Thus, it is needed to extend the class through inheritance and define additional fields for derived classes. The usage of decorator is possible for object extension as well, as decorator helps to avoid inheritance in case of needing to create a larger number of derived classes".

As for the last group of developers with less than 1 year of coding experience, they generally preferred inheritance, as this method is considered the easiest and most convenient.

5 Conclusion

The study on programmers' preferences in object-oriented programming has yielded valuable insights into the popularity and usage of various object extension methods. The findings suggest that inheritance remains the most favored approach among developers, with a significant lead over other methods. Speaking regarding the programmers with more than 10 years of coding experience, they predominantly choose inheritance, composition, and static methods, with inheritance being the clear favorite. Decorator, despite its benefits, is not as widely adopted in this group.

The study highlights the enduring appeal of inheritance, which is often cited as being familiar, obvious, and simple, allowing code to be understood by a larger number of developers. However, it also suggests that as programmers gain more experience, they may become more open to alternative approaches like decorator and composition, which offer benefits in terms of flexibility and maintainability.

References

[1] G. Antoniol et al. “Object-oriented design patterns recovery”. In: *The Journal of Systems and Software* 59 (2001), pp. 181–196.

[2] Myung-Hoon Chung. “Science Code .Net: Object-oriented programming for science”. In: *Science of Computer Programming* 71.3 (2008), pp. 242–247.

[3] William Flageol et al. “A mapping study of language features improving object-oriented design patterns”. In: *Information and Software Technology* 160 (2023), pp. 107–222.

[4] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[5] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[6] G.A. Stepanov. “Object-oriented programming”. In: *CURRENT SCIENTIFIC RESEARCH : collection of articles of the XII International Scientific and Practical Conference* 4.2 (2023), pp. 98–100.

[7] V.A. Volkov. “Analysis of the importance of mandatory use of design patterns in the process of software design and development”. In: *Integration of sciences* 8.4 (2017), pp. 51–53.