UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

COS700 Research

Design Pattern Metaprogramming Foundations in Rust

Abstract Factory and Visitor

Student number: u19239395

Supervisor:
Dr. Linda Marshall

??? 2020

Abstract

Keywords:

1 Introduction

2 Design Patterns

Software design focuses on the design and implementation of software to solve a particular problem [iee09, SJB15]. There should be no surprise to see some problems repeating themselves with time [GY11]. The solutions to these problems are the same each time. But a novice designer facing any of these repeated problems for the first time will try to solve them from first principles [GHJV94, Son98]. When the solution proves flawed or misunderstood some weeks later, a small improvement will be made to the solution [Zhu05, iee09]. These improvements are repeated until all the flaws are removed from the solution [Ste15, SJB15].

On the other hand, seasoned designers create good designs from their own or their colleagues' past experiences [Son98]. These solutions are easy to find in mature libraries and projects [GHJV94]. However, novices are unlikely to get exposure to these projects [Zhu05] or are just overwhelmed by their size [HSG18]. Having exposure to these projects will allow novices to jump to the good design directly. Thus, saving time on the iteration process [SJB15].

But rather than taking novices to the projects, it might be possible to take the designs to the novices [CK03]. This is exactly what happened in the 90s. The Gang of Four took some of the repeated designs in projects and documented them in "Design Patterns: Elements of Reusable Object-Oriented Software" [GHJV94].

Each pattern is documented with a name, the problem it is solving, the solution and consequences of using it. Thus, each pattern is an explicit specification for the solution's design while the name becomes a vocabulary encapsulating the specification [GHJV94, BJ12]. This report will focus on only two of the patterns presented by the Gang of Four: Abstract Factory and Visitor.

2.1 Abstract Factory

During the instantiation of classes, four independent sets of problems might exist. The Abstract Factory design pattern proposes to be a solution to these four problems. [GHJV94]

2.1.1 Problems

The first problem is when the instantiation and representation of classes need to be separate from the application code. Keeping data structures in a standard library and not the application code is an example of the first problem. It can be argued that using Abstract Factory for this problem might be over-engineering the solution [Ker05].

A second problem is the reverse of the first. When a designer wants to create a library of objects but only expose their interface and not their implementation. In a GUI library, only exposing the operations on a button and not the fact that the button is blue or glossy is an example of hiding the implementation.

The designer wanting to have a family of related objects to be used together is the third possible problem. Forcing the glossy button to appear with the

glossy scroller is an example of wanting the object families together.

Lastly, wanting to swap a family of products for another family of products is the fourth possible problem. This is, swapping all the glossy GUI items to the flat blue items for the entire application by changing one line will be nice.

For this report, we will only focus on problems two to fourth.

2.1.2 Solution

Problems two and four requires each product to have an abstract definition - called *Abstract Product*. Doing so will hide the implementation for problem two - nothing that follows is really needed for problem two. Meanwhile, for problem four, all the application code will operate against the interface for a button and scroller and not their concrete implementations. Thus, swapping from the glossy to the blue elements will not require any additional code changes at the method calls.

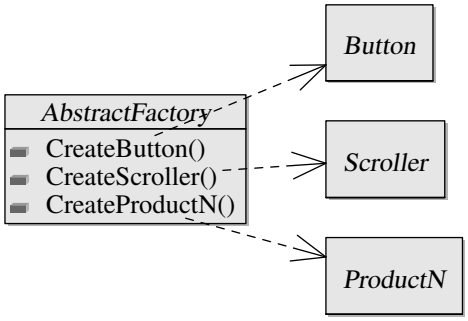

Figure 1: Interfaces needed for Abstract Factory

Problems three and four both need to control the instantiation of a family of products. Therefore, a class dedicated to products creation will be needed. Problem four needs this class to be abstract to swap one family for another - hence why this class is called *Abstract Factory*.

So far, we have the design in figure 1.

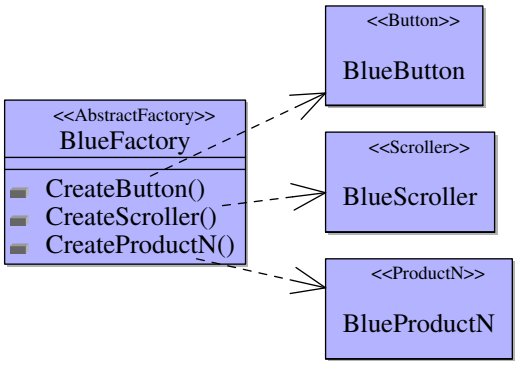

Figure 2: Concrete BlueFactory

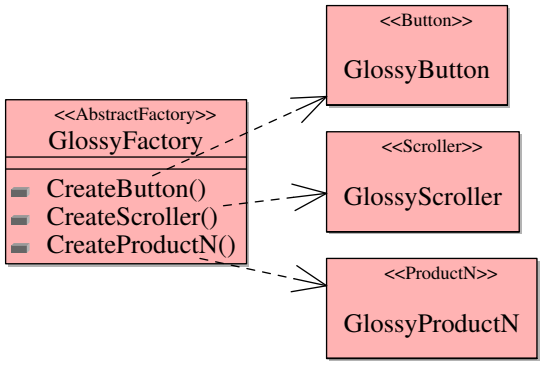

Figure 3: Concrete GlossyFactory

Problem three does not need *AbstractFactory* to be abstract. Only problem four needs the abstraction to be able to swap one family for another. Figure 2 on the previous page shows how the concrete blue GUI family maps to all the abstractions. The same mapping can be seen for a glossy family in figure 3. Since both concrete designs have the same interface, swapping the one for the other is non-trivial.

2.1.3 Consequences

Thus, the *Abstract Factory* pattern makes it easy to group a family of related products and swap one family for another. By having client code only work against the abstractions, the *Abstract Factory* pattern also isolated the concrete implementations from the client code.

However, adding a new abstract product to the family creates a drawback. Each concrete family will have to add its own concrete form of the product too. Thus, the number of classes needing to change is $n + 1$, where n is the number of families [BJ12].

2.2 Visitor

Performing an operation on a set of objects can be quite difficult. The *Visitor* design pattern proposes a solution to three problems [GHJV94].

2.2.1 Problems

For the first problem, imagine classes all with different interfaces. But, an operation needs to be performed against each concrete class. For example, a button and a scroller have different interfaces. However, both have to be drawn. Alternatively, a need might exist to read both aloud for the screen-reader.

Doing unrelated operations with the classes is a second problem. For a study, a company might want to know the average screen surface area of its GUI elements. This is unrelated to a GUI library. Adding surface area methods to GUI classes will pollute the classes.

Lastly, the classes may rarely change as a third problem, but the operations performed on them change often. Coming back to the study, a week later finding

the most common element color might be needed. No new elements were added to the GUI library. Only the need for a new operation exists.

2.2.2 Solution

The solution is to look outside the classes. Thus, creating a new class which knows how to perform only a single operation. The new class will need to visit each of the classes in the problem space. This class is called *Visitor* and solves problems one and two.

However, problem three adds a new dimension. Creating a new operation means creating a new visitor type. Since they are both visiting the same classes, they are both the same in an abstract sense. It is only their implementations that differ. Thus, having an *Abstract Visitor* to represent both is needed.

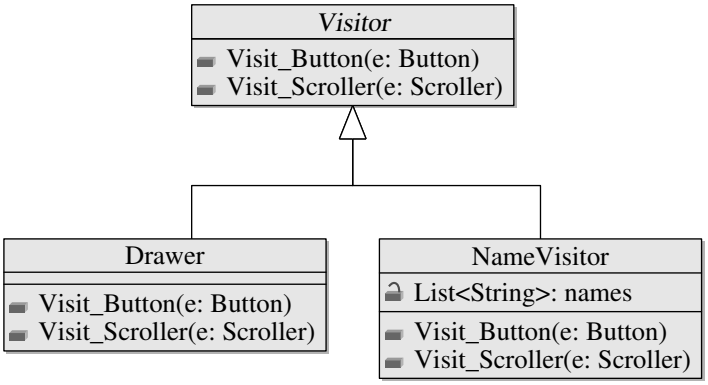

Figure 4: Interfaces needed for Abstract Visitor

Abstract Visitor will have a method for each class it needs to visit as seen in figure 4. Requiring the client to remember the method corresponding to each class will not be ideal when the classes reach more than 30. It is also not ideal for generic pieces of code since the method names are not the same.

To solve this, each class has a method to *accept* a visitor as seen in figure 5 on the following page. This method calls for another abstraction called *Element* with the *accept* method. In the *accept* method, each class can call the visitor operation corresponding to it.

2.2.3 Consequences

New operations (*Visitors*) can easily be added without touching the classes. Related operations are now also isolated to each visitor. Thus the classes are not polluted with unrelated methods. Visitors also store the state information they need rather than passing it to each function as seen in *NameVisitor*.

However, there are two problems. First, adding a new class means updating all the visitors with a method for it. Second, it is assumed that each class exposes enough information through its public interface for visitors to perform their needed operations.

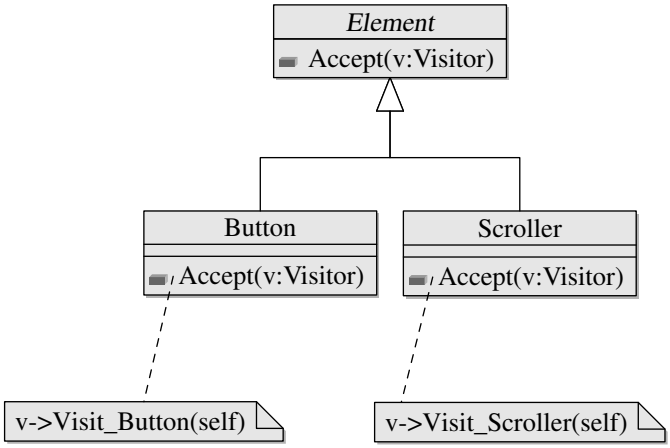

Figure 5: Accept on elements to visit

3 Reporting

4 Conclusion

References

[BJ12] Aleksandar Bulajic and Slobodan Jovanovic. An approach to reducing complexity in abstract factory design pattern. *Journal of Emerging Trends in Computing and Information Sciences*, 3(10), 2012.

[CK03] David Carrington and S-K Kim. Teaching software design with open source software. In *33rd Annual Frontiers in Education, 2003. FIE 2003.*, volume 3, pages S1C–9. IEEE, 2003.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[GY11] Davoud Keshvari Ghourbanpour and Mohhamd Hossien Yektaie. Towards pattern-based refactoring: Abstract factory. *International Journal of Advanced Research in Computer Science*, 2(3), 2011.

[HSG18] Zhewei Hu, Yang Song, and Edward F Gehringer. Open-source software in class: students' common mistakes. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training*, pages 40–48, 2018.

[iee09] Ieee standard for information technology–systems design–software design descriptions. *IEEE STD 1016-2009*, pages 3–4, 2009.

[Ker05] Joshua Kerievsky. *Refactoring to patterns*. Pearson Deutschland GmbH, 2005.

[SJB15] John W Satzinger, Robert B Jackson, and Stephen D Burd. *Systems analysis and design in a changing world*, chapter 1, 2, 13, pages 5, 44, 428. Cengage learning, 2015.

[Son98] Sabine Sonnentag. Expertise in professional software design: A process study. *Journal of applied psychology*, 83(5):703, 1998.

[Ste15] Rod Stephens. *Beginning software engineering*, chapter 13, page 284. John Wiley & Sons, 2015.

[Zhu05] Hong Zhu. *Software design methodology: From principles to architectural styles*. Elsevier, 2005.