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 designing and implementing 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 designs focus not only on immediate development but also on the development needed during maintenance [Ker05, GHJV94]. These solutions are easy to find in mature libraries and projects [GHJV94]. Unfortunately, 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. This saves 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. Gamma et al., which the rest of this report will refer to as the Gang of Four (GoF), 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 the 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]. The GoF also groups the patterns into 3 categories: creational, structural, and behavioral. This report will focus on one creational pattern - *Abstract Factory* - and one behavioral pattern - *Visitor*. No Structural patterns will be discussed. Since the *Factory* pattern can be used to implement the *Abstract Factory* pattern [Nyk12, GHJV94], this section will cover *Factory* too. A discussion of *Factory*, *Abstract Factory*, and *Visitor* follows.

2.1 Factory

When an object is created, an isolated function/method may not care which concrete version gets created. It may only care about an abstract definition of the object to perform its duties. The Factory design pattern proposes to solve three variants of this problem. [GHJV94]

2.1.1 Problems

The first problem is when a function does not have enough information to determine the concrete object it should create. The method/function only knows the abstract object it wants. An example of this is the button on a confirmation dialog. The abstract confirmation process only needs to create a button.

Whether this is a blue button used during saving or a glossy button used when installing is not the abstract confirmation process's concern.

Problem two joins in with problem one. Needing to create more than one button means logic to decide on a button to create. Duplicating this logic at each instantiation will make maintenance hard. Imagine the blue button is decommissioned as part of a new facelift in favor of a red button. Updating every line instantiating a blue button might just kill the developer of boredom.

Having a superclass delegate the creation responsibility to a subclass is a third problem. Since all dialogs follow the same process, the design may call for abstracting the common code into an abstract class. The abstract class will open the dialog, draw the needed elements on it, and destroy it once done. However, the concrete delete dialog and concrete save dialog will need different buttons. The abstract class can then use virtual methods on the concretes to get the correct button for drawing.

2.1.2 Solution

The solution will focus on the first two problems since they relate to Abstract Factory. The first problem calls for an abstraction of the product being created. This will allow the confirmation process to function against an abstract button and not a blue or glossy one.

Problem two calls for the isolation of a button's instantiation from the decision logic. This means another abstract class - called *Factory* - to hold the instantiation of a concrete product. The logic will decide which concrete Factory to use at a later stage. The result is the design in Figure 1. The client code will mostly be working with the interfaces in white.

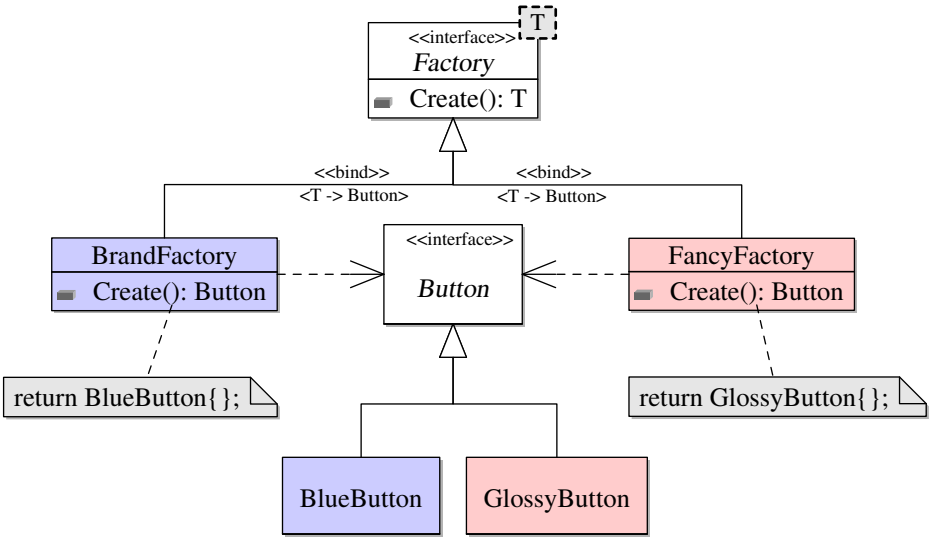

Figure 1: Factory design

For problem one, functions can create objects from a *Factory<Button>* without worrying if it is working with the brand or fancy factory. The instantiation in *BrandFactory* is the only line needing to swap to a red button for problem

two. The single logic decision point will be the only client code containing the Brand and Fancy factory classes. No client code will contain the blue or glossy button classes.

2.1.3 Consequences

The client code is no longer bound to a concrete button. It now just works with an abstract button. Also, the logic to choose a factory appears once in the code.

However, this solution does require a new factory to be created for each button type. If a new transparent button is needed next week, then a new opacity factory will be needed. Maintenance is still easy since only the single logic point needs to be updated to introduce the new factory. The rest of the client code still does not care that it is now working with a transparent button since the white interfaces did not change.

2.2 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]

2.2.1 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. This should remind us of the Factory design just created.

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. Again, reminding us of the Factory design. This time just for more products.

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

2.2.2 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 - effectively solving problem two. 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.

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 it being called *Abstract Factory*. Everything presented so far is the same as Factory's. We now want to create more than one product. Figure 2 shows the Factory design extended to more than one product.

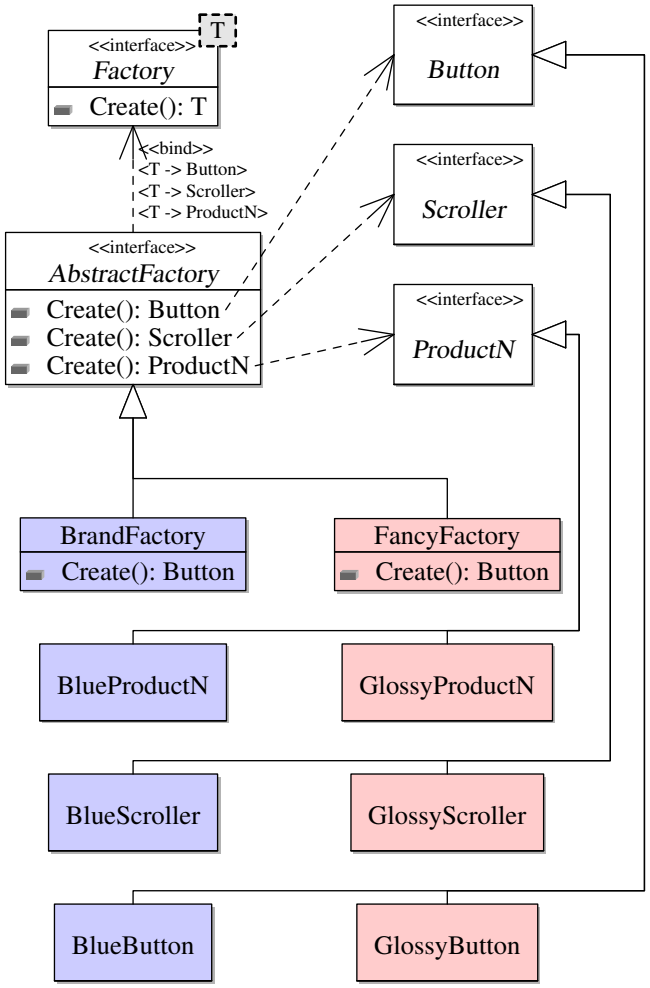

Figure 2: Interfaces needed for Abstract Factory

Problem three does not need *AbstractFactory* to be abstract since it has only one family. Figure 2 shows how the concrete brand GUI family maps to all the abstractions. The same mapping can be seen for a fancy family. Since both concrete designs have the same interfaces, swapping the one for the other is non-trivial since client code will again only operate against the white interfaces.

2.2.3 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. Again, adding a transparent family requires the creation of a transparent factory and each transparent product. However, only the single logic line in the client needs to be updated as a maintenance exercise.

But, adding a new abstract product to the family creates a drawback [GHJV94]. Each concrete family will have to add its own concrete form of the product too. So adding a new window product means creating one abstraction and updating the two families. Thus, the number of classes needing to change is $1 + n$, where n is the number of families [BJ12].

2.3 Visitor

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

2.3.1 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.

2.3.2 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.

Abstract Visitor will have a method for each class it needs to visit as seen in Figure 3. 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.

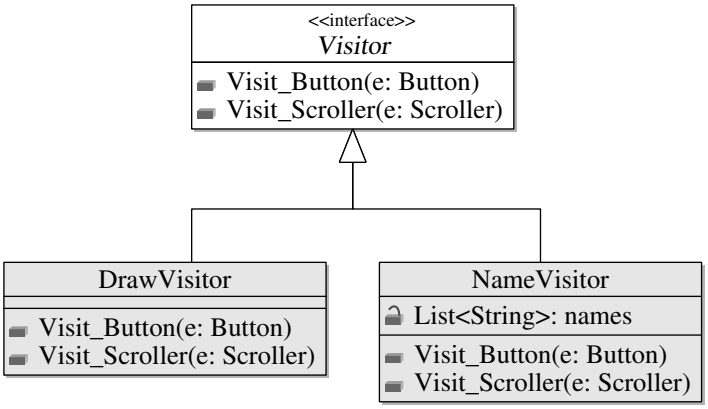

Figure 3: Interfaces needed for Abstract Visitor

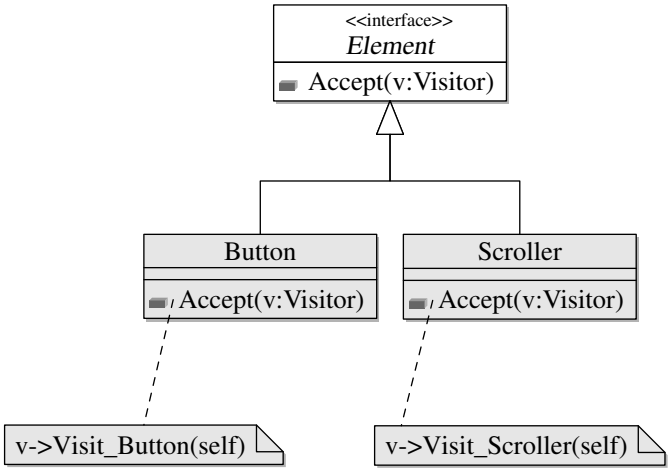

Figure 4: Accept on elements to visit

To solve this, each class has a method to *accept* a visitor as seen in Figure 4. 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.

2.3.3 Consequences

New operations (*Visitors*) can easily be added without touching the classes. Catering for next week's survey means creating a new visitor without touching button or scroller. 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. Thus, adding the window element will

require an update for each visitor to visit it too. Second, it is assumed that each class exposes enough information through its public interface for visitors to perform their needed operations. The scroller may not expose its name. This will leave the name visitor not being able to get the name of scroll elements.

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.

[Nyk12] AA Nykonenko. Creational design patterns in computational linguistics: Factory method, prototype, singleton. *Cybernetics and Systems Analysis*, 48(1):138–145, 2012.

[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.