Super Iterator

A design pattern for Algorithm and Data structure collections

Andre Oboler
Computing Department
Lancaster University
Lancaster, UK
oboler@comp.lancs.ac.uk

Charles Twardy
Information Extraction & Transport Inc.
Arlington, VA
USA
ctwardy@iet.com

David Albrecht
Faculty of Information Technology
Monash University,
Melbourne, Australia
David.Albrecht@infotech.monash.edu.au

Abstract

The Super Iterator pattern, like the standard Iterator pattern, traverses an unknown data structure without exposing that structure. With the standard Iterator pattern, clients must create a different iterator for each new structure, and the object returned must be of the specific type stored in the structure, even when they share a common super class. With the Super Iterator pattern, the object returned is of the common super class, and the iterator itself need not be altered when adding a new subtype with custom data structures. The client, however, must change two lines of code to load and instantiate the new subclass.

Software patterns, Generalisation, Software Architecture, Reuse, Component Based Design

I. Introduction

The Super Iterator design pattern emerged because we needed a more extendable version of the Gang of Four Iterator pattern [1]. We have designed an optimisation library to allocate resources for wilderness Search and Rescue. There are many algorithms in the Operations Research literature, depending on whether the target is moving or fixed, the number of resource types, etc. The algorithms often have their own highly optimised and idiosyncratic data structures [2, 3], but in the end, they all allocate resources to areas. So clients should be able to treat all allocations the same, regardless of how they were created and without knowledge of – or interference with – their management. New algorithms should not alter the rest of the library; neither should we transform the data into some generic structure likely to be suboptimal.

With the Super Iterator pattern, developers wishing to switch to a newly available resource allocation algorithm need change only two lines of their code: first, they include the new algorithm's header file, second, they change one line of code to create an allocation object of the new type instead of the old type. *Once the type is created under this pattern all the rest of their code will work as it did before with no other changes.* This pattern lowers the burden for application developers who wish to integrate the latest research. It also lowers the burden for researchers who can reuse test-harnesses to evaluate new algorithms.

II. Intent

- Avoid coupling the sender of a request to its receiver.
- Allow the sender to iterate over an arbitrary data structure of receiver's subtype as if it were a sequence of super-type, without customized external helper classes.

III. Motivation

While developing our library, we found a new algorithm for multiple resource types [2]. It was highly optimized with a peculiar but very efficient data structure. Still, it did what all these algorithms do – allocated resources to areas. Clients needed to see it as a generic allocation. The traditional Iterator pattern didn't work – the client would have to know what kind of iterator to make, and unpack whatever peculiar object it pointed to. It was clear we could expect similar problems for most new algorithms.

The "obvious" answer was to make each algorithm convert or copy its objects to a generic allocation table. But the idea was repugnant – the data was already there, efficiently packed. Why copy it? Furthermore, modifiable allocations presented a synchronization problem, and overhead. The approach also complicated things for the developer.

We needed abstraction. The Super Iterator pattern uses a single set of iterators for all subclasses and protects the data from alteration. Therefore clients can switch algorithms just by changing the initial call type. The iterators they previously coded will work as before, even though the new algorithm may consider many more factors and create a much more complicated structure.

To make it work, each algorithm makes available a set of helper methods that befriend the generic iterators. In effect this moves the mechanics of the GoF helper classes inside the algorithms, hiding them from the client.

IV. Applicability

Use the Super Iterator pattern:

- To view data as an instance of the generic or super type without storing it as such
- To provide a uniform interface for traversing various aggregate data structures without creating specialized iterators for each structure
- To support multiple traversals of aggregate objects while only storing one node object per iterator

V. Structure and participants

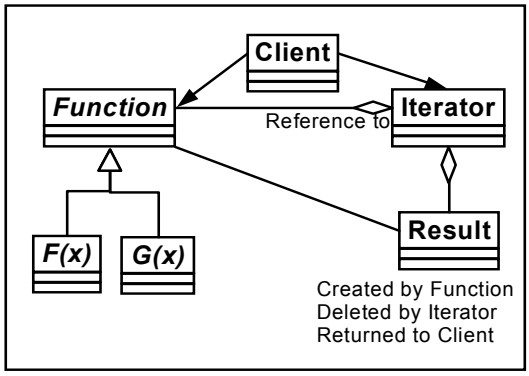

Figure 1. The Patterns stucture

- Function – defines the class interface
- F(x), G(x) – specific functions with concrete data structures and algorithms. Also defines how to traverse that data structure.
- Result – abstractly, all Functions are aggregates of objects of this type
- Iterator – defines a uniform interface through which the client program can interact with the Functions
- Client – the client program

VI. Collaborations

The client program creates a Function of choice e.g. F(x) in Figure 1, which manipulates the data x and stores the results in a specialist internal data type. The iterator presents F(x) as if it were a collection of Result objects. The iterator causes the function to create a single Result object containing the appropriate value for the "current" node. The Result class is not the native class that F(x) uses to store the data, so the values are copied to this standard format. The iterator manages the Result class, deleting it and freeing the memory when needed. The client program can access the Result using the iterator's dereference method.

A new function H(x) can be added and once the client creates H(x) instead of F(x), the rest of the code should work without alteration.

VII. Consequences

The full functionality of Iterator is retained including:

- Variation in the traversal order of the aggregate
- Simplifies the aggregate interface
- Multiple traversals on the aggregate

Additionally,

- Data is returned in a common form by the Iterator regardless of the way it is stored in the aggregate
- Only one generic Iterator class is needed (not a specialist one per data type)
- New function classes can easily be added

VIII. Known Uses

This pattern has proved useful for the specific problem of Search and Rescue and has been implemented in the SORAL library [4]. A simplified toy problem based on the SORAL implementation with the use of lists as F(x) and maps as G(x) is also available.

This pattern should be useful for data intensive work developed in an Object Oriented paradigm but seeking the flexibility and efficiency of custom data structures and manipulation algorithms.

IX. Related patterns

This design pattern extends Iterator and makes some use of Abstract Factory and Bridge.

X. References:

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Mass.: Addison-Wesley, 1995.

[2] A. D. Washburn, "Finite method for a nonlinear allocation problem," *Journal of Optimization Theory and Applications*, vol. 85, pp. 1573—2878, 1995

[3] A. Charnes and W. W. Cooper, "The Theory of Search: Optimum Distribution of Search Effort," *Management Science*, vol. 5, pp. 44–50, 1958

[4] C. R. Twardy, "The SARBayes project," 2002. http://sarbayes.org