This page last changed on Dec 07, 2006 by dswood.

Parking Lot - Visitor

Session Objectives

By the end of this session students should ...

- Understand and be able to implement a Visitor
- Understand *double dispatch*
- Understand when to use preVisit() and postVisit()

Session Overview

Activity
• Lab - Car Count • Lab - Print Name of Attendants and ParkingLots • Lab - Refactor to Visitor • Visitor Pattern • Lab - Indent Printing

Session Notes

Lab - Car Count

Tell the students that you need to know the total number of cars in each parking facility. Ask them to implement.

Things to look for

- Attendant will need to either (1) know about unavailable lots or (2) remember the total number of cars parked in all unavailable lots.
- Tests in both ParkingLotTest and AttendantTest
- assertCarCount()

Lab - Print Name of Attendants and ParkingLots

Tell the students that you would like them to print the names of each attendant and parking lot. Each name should be on a new line.

Discussion
The print implementation will duplicate the traversal logic with car count. Elicit ideas from the students how to avoid this duplication.

If they suggest Strategy, elicit how this situation is different than Strategy:

- has accumulation - Strategy is typically stateless
- multiple types of objects (ParkingLot and Attendant) with different logic for each

Lab - Refactor to Visitor

Ask the students to refactor away the duplication of the tree traversal logic.

There are many directions the students might go with this. Direct them towards the visitor pattern.

Visitor Pattern

Elicit the formal description of the Visitor Pattern:

Title

- Visitor

Problem

- All operations which traverse a tree of objects cause the interface of all of the objects in the structure to handle the traversal. This duplicates for all traversal operations, of which there are often many. Separating the (what can be complicated) traversal logic from the work that needs to be done on the traversal allows you to add new operations to the tree without modifying any of its members. Often the data collection being done does not fall under the job of the object it is being collected from.

Pattern/Solution • Each object in the collection has an accept() method which takes the visitor • accept() on collections calls visit(this) and accept(for each child) • Visitor has visit() method for each class type as necessary. (ie. visit(ParkingLot) & visit(Attendant)) ◦ This is an example of *double dispatch*: A mechanism that dispatches a method call to different concrete methods depending on the runtime types of the method parameters. This is more than method overloading in which the object on which the method is being called is determined at runtime, but the types of the method parameters are fixed. • May optionally have a preVisit() and/or postVisit() if you need to do work before and after visiting children • Common to use in conjunction with Composite	!Parking Lot Visitor.png!

- Sometimes hide the visitor behind a method call - let the method build the visitor, collect the data, and then have the method return the result from the visitor

Examples

- Since commonly used with Composite, composite trees may be a good example. Parse Trees (DOM, AST, etc.)

Benefits/Liabilities

- If the code in subclasses in your tree changes or is added/removed frequently, it will cause maintenance pain in having to modify all of the visitors. This means that Visitor is best for structures with a fairly static class heirarchy.
- Cleanly separates traversal logic from non-traversal logic (ie. separates tree-walk algorithm and removes need for carCount() & print() methods on the Attendant and ParkingLot interfaces)
- Can make design and tracing/debugging more complex, so don't use it to replace all iteration over loops.
- Again breaking encapsulation a bit, but in a controlled manner. It is like a neighbor knocking on the door and you 'accept' them in. You can then only allow them into the dining room and not into the kitchen.
- May need preVisit() and postVisit() methods on the Visitor to keep track of location in a hierarchy

Refactoring steps

- Implement accept() method on each business object
- Create visitor interface with a visit() method for each business object
- Create an implementation of the visitor for each usage you're replacing
- Change the public methods to use the new visitor internally

Lab - Indent Printing

Ask the students to indent the printing so that deeper nodes in the tree are more indenteded.

Things to look for

- Rename visit(Attendant) --> preVisit(Attendant) which also indents
- Add postVisit(Attendant) after the Attendant's children have been visited, which removes indenting
- Point out the preVisit() / postVisit() pattern allows you to set up conditions when visiting the subtree under a tree, and then tear down the conditions afterwards.
- There should not be a preVisit(ParkingLot) or postVisit(ParkingLot).
- Could take advantage of default implementations for preVisit(Attendant) and postVisit(Attendant) by changing the visitor interface to an abstract class. That way visitor subclasses like ParkingLot that do not care about preVisit(Attendant) or postVisit(Attendant) do not need to implement them.