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

Parking Lot - Composite

Session Objectives	Session Overview
By the end of this session students should ... • Understand the concept of a collection acting like an individual item • Recognize situations in which a Composite may be appropriate • Understand and be able to implement a Composite • Feel comfortable undertaking a major refactoring while keeping their tests green	**Activity** • Lab - Filling lots by least full • Lab - Attendants with Employees • Lab - Additional Tests • Formally describe the Composite pattern

Lab - Filling lots by least full

Ask the students to have the attendant fill lots by least full (by percentage, but make them ask for this).

Discussion
How can we test this (without pillaging the Lot's data)?

- Add two lots to an attendant.
- Have the attendant park a couple of cars.
- Use your friendly assertLotFull method on the lot (which tries to park again and catches the LotFullException)

Things to look for

- Comparator on Lot that orders by percent full
- Use this Comparator and the Collections.min() method to find the proper Lot to use

Lab - Attendants with Employees

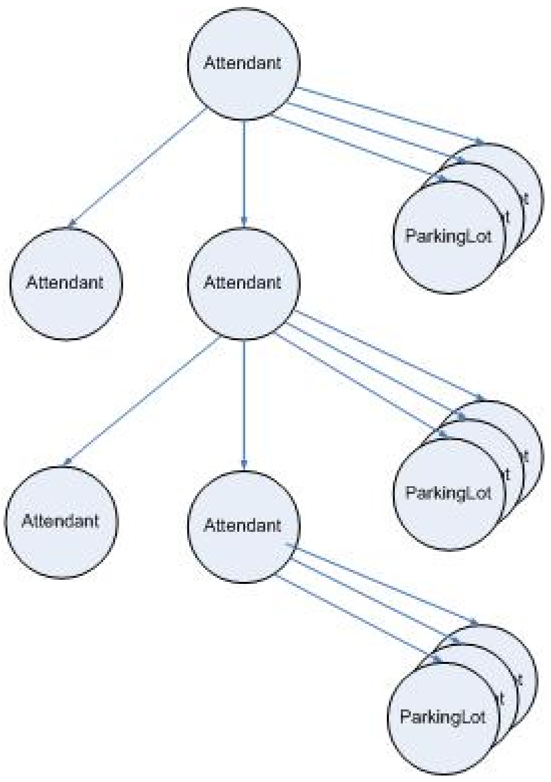
Baseline all students at this point. Hold on to that baseline, you'll probably need it again.

Ask the students to allow Attendants to hire other Attendants to do some work for them. (ie. I'm managing some lots myself, but when I take those into account I want to ask my cousin Vinny about his as well.) Give pairs 5 minutes for design discussion. Then discuss as class. Ask one student to draw the Attendant - Parking Lot relationship on the board (see picture). The students should have decided to add a collection of Attendants to Attendant in addition to the collection of ParkingLots. The first test must be simple - one employee with one lot. Emphasize that the students should keep existing tests green as they modify the code. It will make their lives easier.

Ask the students to implement.

<u>Things to look for</u>

- All existing tests and the new test working
- Collection of Attendants on Attendant
- Best lot by percent full still has to be taken into account
- if-else statements in park() method
- is it valid to add a listener to Attendant? It would only serve to "decouple Attendant from itself" at this point.
- Comparator on Attendant allowed us to get rid of some null checks
- Two objects are becoming more and more similar. At what point do we recognize a possible shared interface and extract it?
- Watch for potential rounding issues with percentage full



<u>Discussion</u>
As the students move towards a multi-level tree with multiple parkinglots and attendents, there will be lots of duplication between Attendant and ParkingLot. Is there really a difference between an Attendant and a ParkingLot? Do they not have the same kind of methods doing the same thing? Expect some argument here, but stick to the code doing the same thing as far as the invoker is concerned (internals are encapsulated, after all). The similarity of the behavior of the API is the code talking to you.

Introduce the Composite pattern (aka Thing-Group-Thing in pre-GOF days). Tie it to tree structures - nodes and leaves.
Elicit some examples: Governments, Corporations, Swing Components, parsing trees (XML/DOM), Constraints typical in business (this and this, but not if that)

Some things to discuss:

- The concept of a collection acting like an individual item

- Composite is not for all collections, only for collections of collections (Thing - Group - Thing)
- The trade-offs between using an abstract class and an interface
- The power of existing tests making it safe to undertake the major refactoring to composite. The value of keeping tests running while refactoring.

Ask the students to implement Composite.

Lab - Additional Tests

Show how easy it is (in just the test setup) to create arbitrary hierarchies that still work. No new code required!

Test cascading notification. If you have 3 levels of attendants that are full, and the lot of the bottom attendant frees up, the top attendant should now be able to park.

Formally describe the Composite pattern

Discuss whether building the hierarchy is part of the Composite API. I usually point out that the building of a hierarchy is a different job than the using of a hierarchy (Factory pattern calls that out).

Then, run to the GOF book for all the interesting variations. Cite examples where there are multiple types of aggregations (we used a Parallel and Serial service aggregations to model work flow at OnStar) and multiple types of leaves.

Title

- Composite (Thing-Group-Thing)

Problem

- Individual objects and collections of those objects have very similar functionality, and thus there is code duplication across those classes. Having to distinguish between the object and its collection makes the code complex. This pattern allows you to treat individual objects and collections of those objects uniformly.

Pattern/Solution • May start out with just an interface. Normally will end up with an abstract class that contains common functionality. • Often will have a definite leaf type that cannot handle child objects • For uniformity, the child-handling methods will often be declared on the abstract class. When trying to use them on the leaf node, they will either do nothing or throw exceptions. • Occasionally, if performance or clarity	!Parking Lot Composite.png!

requires, nodes in a composite structure will also have ways of accessing their parent object in the tree.	
• Clients generally don't know (and shouldn't care) whether they are dealing with a leaf or collection	
Examples	
• Swing Components	
• parsing trees (XML/DOM)	
• Governments	
• Corporations	
• Constraints typical in business (this and this, but not if that).	

Benefit/Liabilities

- Cloning can be difficult. How deep do you clone?
- Can add methods to the leaf for child handling that make no sense
- Makes it harder to restrict the components. You can no longer use compile-time checking of types to restrict what can be added where. You have to use runtime checks instead.

Refactoring steps

1. Move to a common interface between the object and collection of objects
2. Extract a superclass
3. Pull common methods up
4. Start with methods having the least dependencies and work your way to the more difficult parts of the relationship