Observer

• One to many relationship
  • The many need to know changes in “one” immediately

• Example
  • Points & Shapes
  • Map & location-based services
  • A game character & other game components
  • ...

Example

• If a person/subject changes its status, how to let all his “subscriber” knows?
  • What to do when there is only one subscriber?
  • What to do when there are multiple subscribers of different types?
  • What if new subscribers are added?
  • How to make the code easy to maintain and extend?
Class diagram

```
for each view in views
  v.update()
```

```
model.getState();
```
Example (location, location-related service)

• “location” would be the *Subject* in previous slide
• “observer” would be the superclass of all the sub-classes that try to update themselves based on the location information
The benefit of observer pattern

• When new types of observers are added, the prototype and implementation of the subject class doesn’t need any changes.
Other things to pay attention

• Don’t forget the subscribing and unsubscribing methods
• Pull notification vs push notification
• What if I want to delete a subject
• Can an observer subscribe multiple subjects?
Composite pattern

• Tree hierarchy
• How do you build a tree?
How to build a tree and traverse it?

struct node{
    struct node* left;
    struct node* right;
    int val;
    int sum(){
        ...
    }
}

How to differentiate leaves and others?

```c
struct leaf{
    int val;
    int sum(){ return val;}
}
```
How to accommodate different types of internal nodes?

• Examples
  • struct node or struct leaf?
  • Book
  • Graphics
Class diagram

```
// Container functionality
// for each element
for (i = 0; i < elements.length; i++) { elements[i].doThis(); }
```
Apply composite pattern to tree

- “Leaf” in previous slide is tree leaf
- “Composite” in previous slide is non-leaf nodes in a tree
Interpreter

• What is an interpreter
  • Language, compiler

• Example
  • Boolean expression
    • Abstract syntax tree

a && b || !c

A parser will turn this into an abstract syntax tree, and then an interpreter will evaluate the tree. How to write a program to do the tree-based evaluation?
How to do addition & subtraction

• How to represent an addition expression?
  • Constant + Constant
  • Constant + Constant + Constant

• How to represent a subtraction expression?
How to do addition & subtraction

• How to represent an addition expression?
  • Tree is a good form

• How to represent a subtraction expression?
  • Tree

• The challenge:
  • Any node in the above tree could be a constant, an addition expression, or a subtraction expression, etc.
Expression
+ evaluation(): int

Constant
+ value: int
+ evaluation(): int

int Constant::evaluation()
{
    return value;
}

Addition
+ evaluation(): int

int Addition::evaluation()
{
    return operands[0].evaluation()
    + operands[1].evaluation();
}

Negation
+ evaluation(): int

int Negation::evaluation()
{
    return -operand.evaluation();
}
Class diagram

Client

AbstractExpression
  +solve(inout Context)

Context

TerminalExpression

CompoundExpression
  +solve(inout Context)

Perform "parent" functionality then delegate to each "child" element
"Context" is data structure for holding input and output