Design patterns
What are design patterns

• Solutions to specific problems in OO software design
• 23 patterns in 3 categories
  • Creational
  • Structural
    • Composite
    • …
  • Behavioral
    • Observer
    • Interpreter
    • …
Why are we studying them?
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
// Container functionality:
// for each element
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.
Class diagram

- Client
- AbstractExpression
  + solve(inout Context)
- Context
- TerminalExpression
- CompoundExpression
  + solve(inout Context)

Additional annotation:
Perform "parent" functionality then delegate to each "child" element. "Context" is data structure for holding input and output.
Strategy Design Pattern

Classes centered on operations, instead of data
Strategy

• Multiple variants of one algorithm
• Different types of objects only differing in behavior
• The key part of a class is its method, NOT its data
  • Example: printer, sorter, comparator
  • The method works for multiple data types
Example

• Printers
  • Various font size, indentation, capitalization
Class diagram
-- encapsulate algorithms into class

Program to an interface, not an implementation.

Client --> Abstraction
+doSomething()

Open for extension, closed for modification.

ImplementationOne
+doSomething()

ImplementationTwo
+doSomething()
Alternative solutions

• If in C

• Super-class on the data side

• Template in C++
Other examples

• Different sorting
• Different rendering
• …
Template

• Provide a skeleton for similar algorithms
  • The key of the class is still operation, not data

• Example
Class diagram

```
FrameworkClass
+templateMethod()
+stepOne()
+stepTwo()
+stepThree()

stepOne();
stepTwo();
stepThree();
```

```
ApplicationClassOne
+stepTwo()

ApplicationClassTwo
+stepTwo()
```
Visitor

• How to add a class of operations for a variety of data classes?

• Example
  • Different operations for AST nodes
  • Different operations for Person (Female, Male)
Visitor

• Two class hierarchies: data & operations

• What is it good at?
  • If you add operations (Visitor classes), the interface of the Element classes remains unchanged

• What is it bad at?
  • If you add new Element sub-class, significant changes are needed for the Visitor side

• Double-dispatch
  • Imagine two dimensions of a function call
    • The exact algorithm
    • The type of data this algorithm works on
  • You will get chance to make choice along both dimensions dynamically, using visitor pattern
Creational design patterns
Factory Method

• Lets a class defer instantiation to subclasses
  • No need to decide which subclass I want to use statically

• Example
  • Date (US style, Europe style, Chinese style, ...)
  • Window
Class diagram

Factory design pattern is somewhat similar with Strategy design pattern.

```
Product

ConcreteProduct

Creator
+FactoryMethod()
+AnOperation()

ConcreteCreator
+FactoryMethod()

product = FactoryMethod()

return new ConcreteProduct
```
When to use factory design pattern?

- The type of the sub-class is determined at run time
- The type changes very infrequently once set
Abstract Factory

• For creating families of related or dependent objects without specifying their concrete classes

• Examples
  • Date, currency, data
  • Window, mouse, scroll bar, …

Abstract factory design pattern is somewhat similar with Visitor design pattern.
Class diagram