C++ & JAVA DESIGN PATTERNS

21st LECTURE

엄현상(Eom, Hyeonsang)
School of Computer Science and Engineering
Seoul National University

Outline

- **C++ Design Patterns**
- **JAVA Design Patterns**

C++ Design Patterns

- Definition
  - Descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context
- Essential Elements
  - Pattern name
  - Problem
  - Solution
  - Consequences
    - Results and trade-off of applying the pattern

텍스트

"Design Patterns: Elements of Reusable Object-Oriented Software,"Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison Wesley, 1995

Visitor: A Design Pattern

- The operation that gets executed depends on both the type of Visitor and the type of Element it visits
- Adds an operation to a class without modifying the class
  - Every class has a virtual method Accept(Visitor& v) reference를 받음
  - For every concrete class S that has Accept, the
  - Visitor has a method VisitS(S* s)
  - An object of class Visitor is passed to the Accept method
  - Accept immediately calls VisitS, passing the this pointer as an argument

각각 아이템에 대한 operation visiter class 인듯
원 클래스 그대로 두고 visitor class에서 각 item에 대한
수행을 따로 visitor 추가해서 사용한 것인듯, (operation 따로)

"Design Patterns: Elements of Reusable Object-Oriented Software,"Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison Wesley, 1995

Visitor and ConcreteVisitor

- Visitor
  - Declares a Visit operation for each class of ConcreteElement in the object structure
- ConcreteVisitor
  - Implements each operation declared by Visitor
  - Each operation implements a fragment of the algorithm defined for the corresponding class of object in the structure
  - ConcreteVisitor provides the context for the algorithm and stores its local state

Element and ConcreteElement

- Element
  - Defines an Accept operation that takes a visitor as an argument
- ConcreteElement
  - Implements an Accept operation that takes a visitor as an argument
- ObjectStructure
  - Can enumerate its elements
  - May provide a high-level interface to allow the visitor to visit its elements
  - May either be a composite or a collection such as a list or a set

원하는 것 정의해서 사용 가능

"Design Patterns: Elements of Reusable Object-Oriented Software,"Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison Wesley, 1995

Visitor Class

operation 분리 -> 새로운 operation 추가 시 용이
but, element 추가시에는 visit element member func 추가 ->
(ELEMENT C) 그러면 원래 visit class가 변경되어야 하는 trade off 존재

```
class Visitor
{
        public:                                  composite element의 pointer
                virtual void VisitElementA(ElementA*);
                virtual void VisitElementB(ElementB*);
                virtual void
                VisitCompositeElement(CompositeElement*);
        protected:
                Visitor();
};
```

"Design Patterns: Elements of Reusable Object-Oriented Software,"Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison Wesley, 1995

ConcreteVisitor Class

```
class ConcreteVisitor : public Visitor
{
        public:
                ConcreteVisitor();
                virtual void VisitElementA(ElementA*);
                virtual void VisitElementB(ElementB*);
                virtual void
                VisitCompositeElement(CompositeElement*);
};
```

"Design Patterns: Elements of Reusable Object-Oriented Software,"Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison Wesley, 1995

Element Class

```
class Element
{
        public:
                virtual ~Element();
                virtual void Accept(Visitor&) = 0;
        protected:
                Element();
};
```

```
class ElementA : public Element
{
        public:
                ElementA();
                virtual void Accept(Visitor& v) {
                v.VisitElementA(this);
                }
};
class ElementB : public Element
{
        public:
                ElementB();
                virtual void Accept(Visitor& v) {
                v.VisitElementB(this);
                }
};
```

"Design Patterns: Elements of Reusable Object-Oriented Software,"Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison Wesley, 1995

CompositeElement Class

```
class CompositeElement : public Element
{
        public:
                virtual void Accept(Visitor&);
        private:
                List<Element*>* _children;
};
void CompositeElement::Accept (Visitor& v)
{
        ListIterator<Element*> i(_children);
        for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Accept(v);
       }
      v.VisitCompositeElement(this);
}
```

"Design Patterns: Elements of Reusable Object-Oriented Software,"Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison Wesley, 1995

How to Use?

```
CompositeElement* e;
Visitor v;
…
e->Accept(v);
```

Or

```
ConcreteVisitor cv;
…
e->Accept(cv);
```

“Design Patterns: Elements of Reusable Object-Oriented Software,”Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison Wesley, 1995

Consequences

- Visitor makes adding new OPs easy
- A Visitor gathers related operations and separates unrelated ones
  - Related behavior is localized in a visitor while unrelated sets are partitioned in subclasses
- Adding new ConcreteElement classes is hard
- Visiting across class hierarchies
- Accumulating state
- Breaking encapsulation

OOP 원칙 중 encapsulation 원칙은 위배라고 말할 수 있다. (엄밀히?

"Design Patterns: Elements of Reusable Object-Oriented Software,"Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison Wesley, 1995

JAVA Design Patterns

- Elegance always pays off
- First make it work, then make it fast
- Remember the “divide and conquer” principle
- Separate the class creator from the class user (*client programmer*) **class user - client programm vs server programm (프로그램 작성은 이 서버 작성 클라이언트 먼저 -> 그 다음 프로그램 작성해야한답**
- When you create a class, attempt to make your names so clear that comments are unnecessary

JAVA Design Patterns Cont'd

- Your analysis and design must produce, at minimum, the classes in your system, their public interfaces, and their relationships to other classes, especially base classes
- Automate everything
- Write the test code first (before you write the class) in order to verify that your class design is complete
- All software design problems can be simplified by introducing an extra level of conceptual indirection
- An indirection should have a meaning

JAVA Design Patterns Cont'd

- **Make classes as atomic as possible**. 복잡하게 만들어졌다는 의미
  Clues to suggest redesign of a class are:
  1) A complicated switch statement: consider using polymorphism
  2) A large number of methods that cover broadly different types of operations: consider using several classes
  3) A large number of member variables that concern broadly different characteristics: consider using several classes

JAVA Design Patterns Cont'd

- Watch for long argument lists
- Don't repeat yourself
- Watch for *switch* statements or chained *if-else* clauses
- From a design standpoint, look for and separate things that change from things that stay the same
- Don't extend fundamental functionality by subclassing
- Less is more

JAVA Design Patterns Cont'd

- Read your classes aloud to make sure they're logical
- When deciding between inheritance and composition, ask if you need to upcast to the base type
- Use data members for variation in value and method overriding for variation in behavior
- Watch for overloading
- Use exception hierarchies
- Sometimes simple aggregation does the job

JAVA Design Patterns Cont'd

- Consider the perspective of the client programmer and the person maintaining the code
- Watch out for "giant object syndrome"
- If you must do something ugly, at least localize the ugliness inside a class
- If you must do something nonportable, make an abstraction for that service and localize it within a class
- Objects should not simply hold some data
- Choose composition first when creating new classes from existing classes
- Use inheritance and method overriding to express differences in behavior, and fields to express variations in state

JAVA Design Patterns Cont'd

- Watch out for *variance*
- Watch out for *limitation* during inheritance
- Use design patterns to eliminate "naked functionality"
- Watch out for "analysis paralysis"
- When you think you've got a good analysis, design, or implementation, do a walkthrough