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

“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)`
  – 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

"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

“Design Patterns: Elements of Reusable Object-Oriented Software,” Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison Wesley, 1995
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
class Visitor
{
    public:
        virtual void VisitElementA(ElementA*);
        virtual void VisitElementB(ElementB*);
        virtual void VisitCompositeElement(CompositeElement*);
    protected:
        Visitor();
};
ConcreteVisitor Class

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


class Element
{
    public:
        virtual ~Element();
        virtual void Accept(Visitor& v) = 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
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

“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*)
- When you create a class, attempt to make your names so clear that comments are unnecessary
• 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.**
  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
• 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
• 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