CS 480
Fall 2015

Mike Lam, Professor

# Visitor Design Pattern

# A brief digression ...

- What are "design patterns" and why are they relevant to compilers?

# A brief digression ...

- What are "design patterns" and why are they relevant to compilers?
  - A reusable "template" or "pattern" that solves a common design problem
    - "Tried and true" solutions
  - Main reference: Design Patterns: Elements of Reusable Object-Oriented Software
    - "Gang of Four:" Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides

# Common Design Patterns

- **Adapter** – Converts one interface into another
- **Factory** – Allows clients to create objects without exactly specifying their concrete class
- **Flyweight** – Manages large numbers of similar objects efficiently via sharing
- **Iterator** – Provides sequential access to a collection without exposing its implementation details
- **Monitor** – Ensures mutually-exclusive access to member variables
- **Null Object** – Prevents null pointer dereferences by providing "default" object
- **Observer** – Track and update multiple dependents automatically on events
- **Singleton** – Provides global access to a single instance object
- **Strategy** – Encapsulate interchangeable algorithms
- **Thread Pool** – Manages allocation of available resources to queued tasks
- **Visitor** – Iterator over a structure (usually a recursive structure)

# Design Patterns

- Pros
  - Faster development
  - More robust code (if implemented properly)
  - More readable code (for those familiar with patterns)
  - Improved maintainability
- Cons
  - Increased abstraction
  - Increased complexity
  - Philosophical: Suggests language deficiencies
    - Solution: Consider using a different language

# Visitor Pattern

- **Visitor**: don't mix data and actions
  - Separates the representation of an object structure from the definition of operations on that structure
  - Keeps data class definitions cleaner
  - Allows the creation of new operations without modifying all data classes
  - Solves a general issue with OO languages
    - Lack of multiple dispatch (choosing a concrete method based on two objects' data types)
    - Less useful in functional languages b/c of functional style and more robust pattern matching

# General Form

- Data: Abstract**Element** (ASTNode)
  - ConcreteElement1 (ASTProgram)
  - ConcreteElement2 (ASTVariable)
  - ConcreteElement3 (ASTFunction)
  - etc.
  - All elements define "Accept()" method that recursively calls "Accept()" on any child nodes
- Actions: Abstract**Visitor** (DefaultASTVisitor)
  - ConcreteVisitor1 (BuildParentLinks)
  - ConcreteVisitor2 (CalculateNodeDepths)
  - ConcreteVisitor3 (StaticAnalysis)
    - (BuildSymbolTables)
    - (TypeCheck)
  - All visitors have "VisitX()" methods for each element type

# Benefits

- Adding new operations is easy
  - Just create a new concrete visitor
  - In our compiler, create a new DefaultASTVisitor subclass
- No wasted space for state in data classes
  - Just maintain state in the visitors
  - In our compiler, we will make a few exceptions for state that is shared across many visitors (e.g., symbol tables)

# Drawbacks

- Adding new data classes is hard
  - This won't matter for us, because our AST types are dictated by the grammar and won't change
- Breaks encapsulation for data members
  - Visitors often need access to all data members
  - This is ok for us, because our data objects are basically just structs anyway (all data is public)

# Minor Modifications

- "Accept()" → "traverse()"
- "Visit()" → "preVisit()" and "postVisit()"
  - preVisit corresponds to a preorder traversal
  - postVisit corresponds to a postorder traversal
- DefaultASTVisitor class
  - Implements ASTVisitor interface
  - Contains empty implementations of all "visit" methods
  - Allows subclasses to define only the visit methods that are relevant