The Visitor Pattern
The Visitor Pattern

- **Intent**
  - Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

- **Motivation**
  - Consider a compiler that parses a program and represents the parsed program as an abstract syntax tree (AST). The AST has many different kinds of nodes, such as Assignment, Variable Reference, and Arithmetic Expression nodes.
  - Operations that one would like to perform on the AST include:
    - Checking that all variables are defined
    - Checking for variables being assigned before they are used
    - Type checking
    - Code generation
    - Pretty printing/formatting
The Visitor Pattern

- Motivation
  - These operations may need to treat each type of node differently
  - One way to do this is to define each operation in the specific node class

![Diagram of the Visitor Pattern]
Motivation

Problems with this approach:

- Adding new operations requires changes to all of the node classes.
- It can be confusing to have such a diverse set of operations in each node class. For example, mixing type-checking code with pretty-printing code can be hard to understand and maintain.

Another solution is to encapsulate a desired operation in a separate object, called a visitor. The visitor object then traverses the elements of the tree. When an tree node "accepts" the visitor, it invokes a method on the visitor that includes the node type as an argument. The visitor will then execute the operation for that node - the operation that used to be in the node class.
The Visitor Pattern

- Motivation
The Visitor Pattern

- Motivation
The Visitor Pattern

- **Applicability**
  
  Use the Visitor pattern in any of the following situations:
  
  - When many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations.
  
  - When the classes defining the object structure rarely change, but you often want to define new operations over the structure. (If the object structure classes change often, then it's probably better to define the operations in those classes.)
  
  - When an object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes.
The Visitor Pattern

- Structure
The Visitor Pattern

- Collaborations
The Visitor Pattern

- Consequences
  - Benefits
    - Adding new operations is easy
    - Related behavior isn't spread over the classes defining the object structure; it's localized in a visitor. Unrelated sets of behavior are partitioned in their own visitor subclasses.
    - Visitors can accumulate state as they visit each element in the object structure. Without a visitor, this state would have to be passed as extra arguments to the operations that perform the traversal.
  - Liabilities
    - Adding new ConcreteElement classes is hard. Each new ConcreteElement gives rise to a new abstract operation on Visitor and a corresponding implementation in every ConcreteVisitor class.
    - The ConcreteElement interface must be powerful enough to let visitors do their job. You may be forced to provide public operations that access an element's internal state, which may compromise its encapsulation.