INF011 – Padrões de Projeto

26 – *Visitor*

Sandro Santos Andrade
sandroandrade@ifba.edu.br

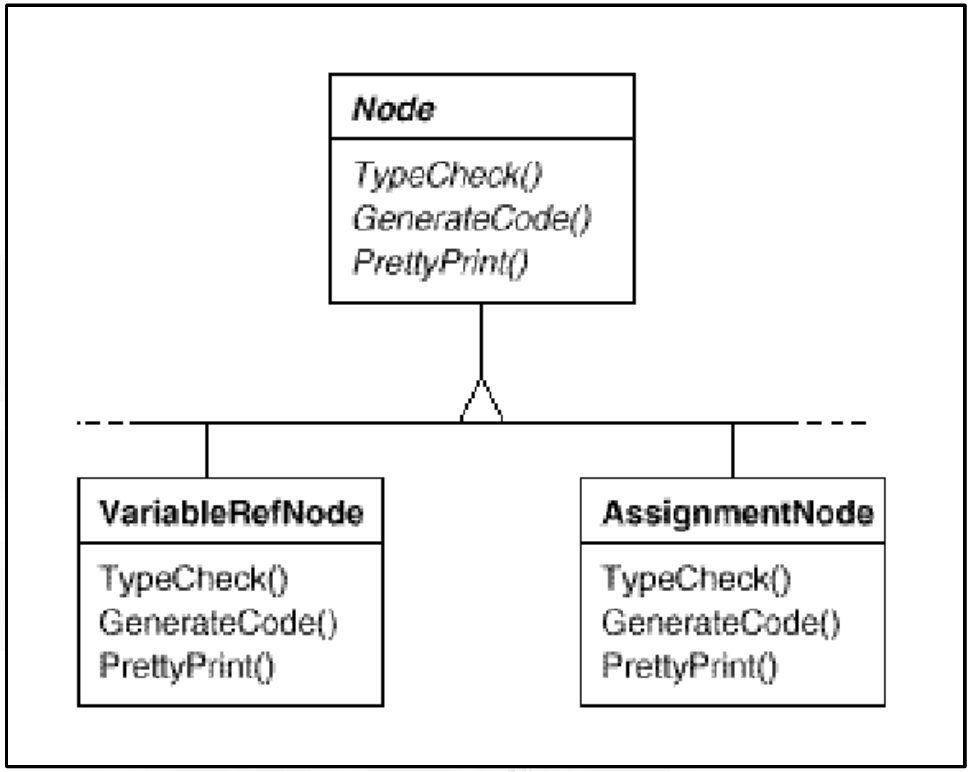
Instituto Federal de Educação, Ciência e Tecnologia da Bahia
Departamento de Tecnologia Eletro-Eletrônica
Graduação Tecnológica em Análise e Desenvolvimento de Sistemas

Visitor

- Propósito:
  - Representar uma operação a ser realizada em elementos de um agregado
  - O *Visitor* permite que você defina uma nova operação sem modificar as classes dos elementos nos quais a operação trabalha
- Motivação:
  - Considere um compilador que representa programas como árvores sintáticas abstratas
  - Diversas operações precisam ser realizadas nesta árvore:
    - Checagem de tipos, otimização, análise de fluxo, busca por variáveis não inicializadas, etc
    - Estas operações tratam nós que representam atribuições de forma diferente de nós que representam expressões, por exemplo

Visitor

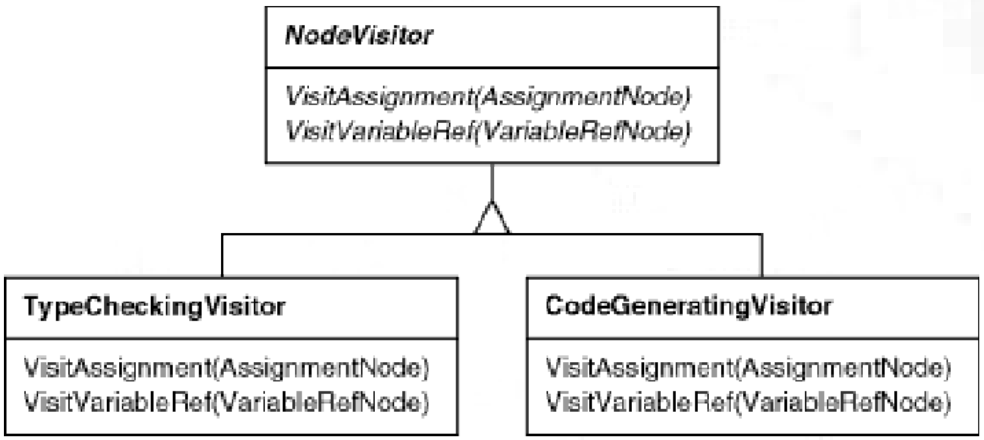
- Motivação:
  - Hierarquia de tipos de nós

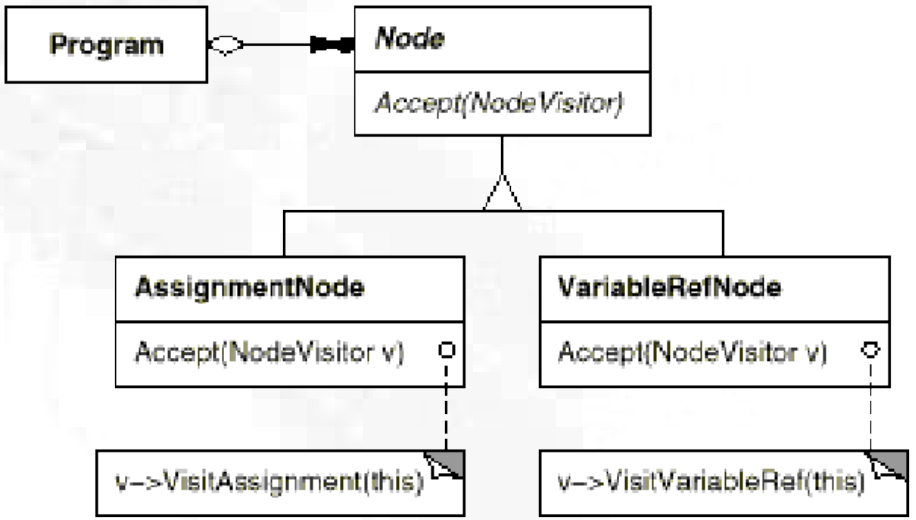


  - Problemas:
    - Distribuir estas operações em várias classes torna o sistema mais difícil de ser compreendido e mantido
    - Adicionar um novo tipo de operação requer a compilação de todas estas classes

Visitor

- Motivação:
  - Solução: encapsular, em uma classe - *Visitor*, os comportamento de uma mesma operação, quando atuando em nós diferentes
  - Introduz-se uma nova hierarquia



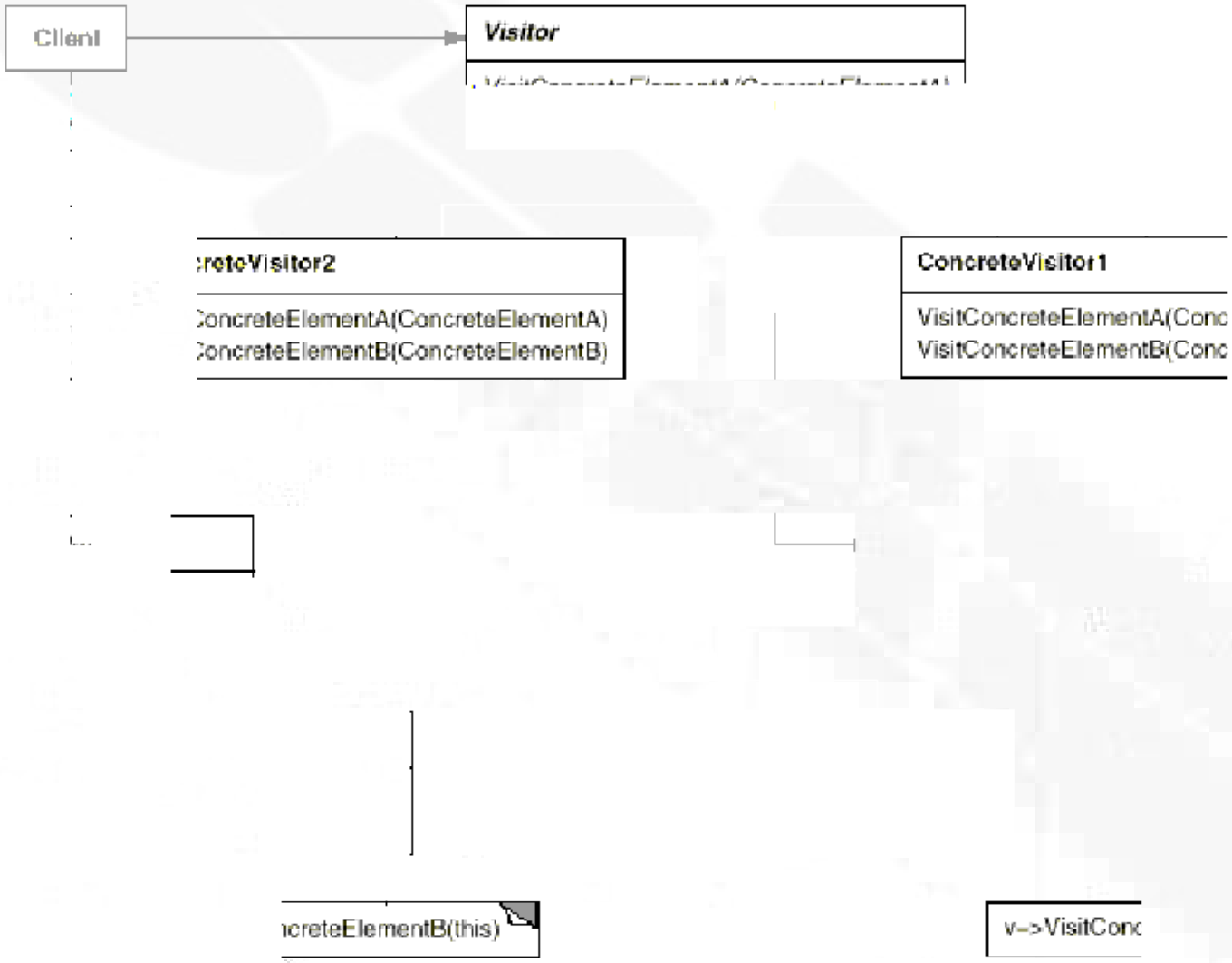


Visitor

- Aplicabilidade:
  - Quando um agregado contém objetos de diversas classes (com diferentes interfaces) e deseja-se realizar operações nestes objetos que dependem das suas classes concretas
  - Quando muitas operações distintas e não-relacionadas precisam ser aplicadas a um agregado e não deseja-se poluir as classes dos objetos do agregado com tais operações
  - Quando as classes que definem os objetos do agregado raramente mudam, porém define-se novas operações com uma certa frequência

Visitor

- Estrutura:



Visitor

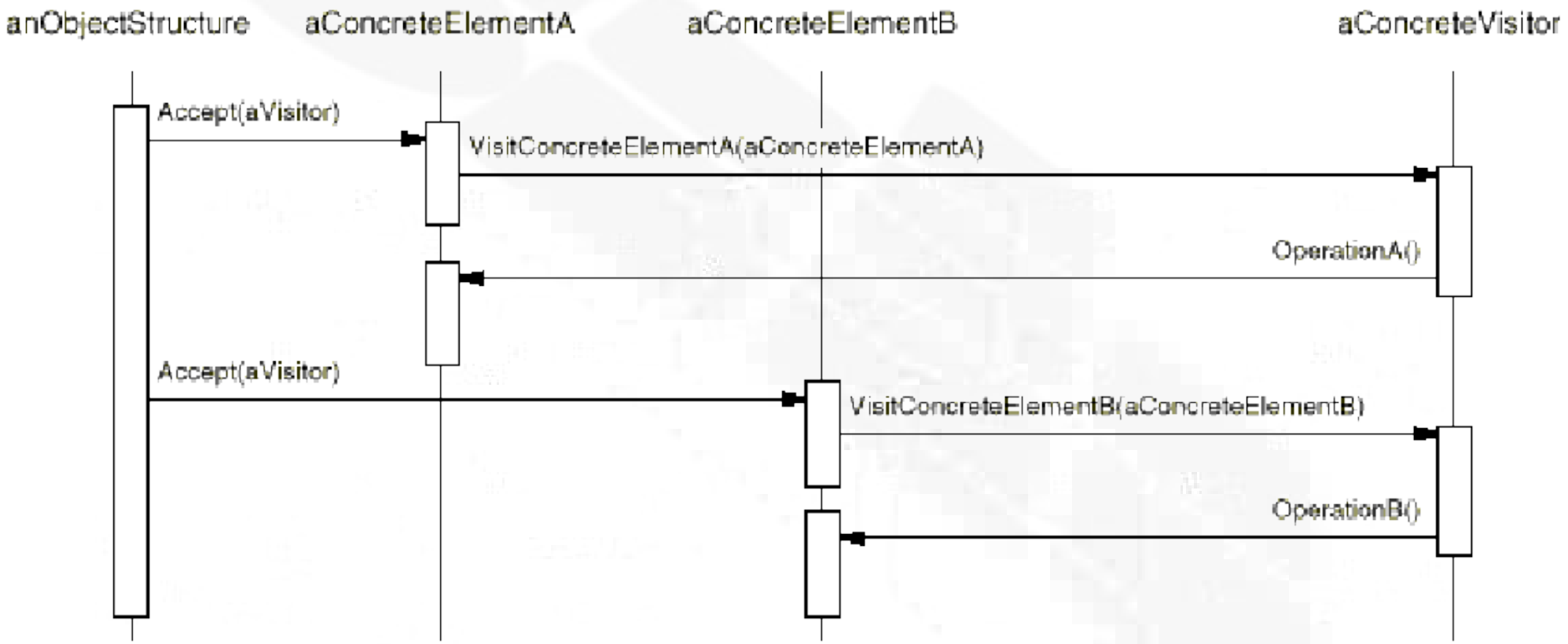
- Participantes:
  - *Visitor* (NodeVisitor):
    - Declara uma operação *visit* para cada classe de *ConcreteElemente* presente no agregado
    - O nome e a assinatura da operação *visit* identifica a classe que solicita a execução do *visit* no *Visitor*
    - Isto permite que o *Visitor* determine a classe concreta do elemento sendo visitado e se comunique diretamente com ele
  - *ConcreteVisitor* (TypeCheckingVisitor):
    - Implementa cadaoperação declarada no *Visitor*
    - Cada operação implementa o fragmento do algoritmo específico da classe em questão
    - O *ConcreteVisitor* disponibiliza o contexto para o algoritmo e geralmente armazena estado local que acumula resultados durante a varredura do agregado

Visitor

- Participantes:
  - *Element* (Node):
    - Declara a operação *accept()* que recebe o *visitor* como parâmetro
  - *ConcreteElement* (AssignmentNode, VariableRefNode):
    - Implementa a operação *accept()*
  - *Agregado* (Program):
    - Pode enumerar seus elementos
    - Pode disponibilizar uma interface de alto nível para permitir que o *Visitor* visite seus elementos
    - Pode ou ser um *Composite* ou uma *collection* como uma lista ou conjunto

Visitor

- Colaborações:



Visitor

- Consequências:
  - O *Visitor* torna fácil a adição de novas operações
  - O *Visitor* agrupa operações relacionadas e separa as não-relacionadas
  - O *Visitor* torna difícil a adição de novos *ConcreteElements*
  - Visitando hierarquias distintas:
    - O *Iterator* permite realizar operações em objetos de uma única hierarquia

```
template <class Item>
class Iterator {
        // ...
        Item CurrentItem() const;
};
```

Visitor

- Consequências:
  - Com o *Visitor* os objetos visitados não precisam ter um ancestral comum

```
class Visitor {
public:
        // ...
        void VisitMyType(MyType*);
        void VisitYourType(YourType*);
};
```

Visitor

- Consequências:
  - Acumulando estado:
    - *Visitors* podem acumular estado à medida em que visitam os elementos. Sem o *Visitor* este estado seria passado como parâmetro das operações que realizam a varredura ou declarado através de variáveis globais
  - Quebrando o encapsulamento:
    - Ao utilizar o *Visitor* assume-se que a interface do *ConcreteElement* é poderosa o suficiente para permitir que o *Visitor* faça o seu trabalho. Portanto, ele pode exigir que sejam criadas operações públicas que expõem a estrutura interna do objeto

Visitor

- Implementação:
  - *Double Dispatch*:
    - Linguagens orientadas a objetos convencionais apenas o nome da operação e do receptor identificam unicamente o comportamento a ser executado:
      - <referencia>.<metodo>(<params>)
    - Em linguagens com *Double-Dispatch* o comportamento é definido pelo nome da operação e de dois receptores
    - No caso do *Visitor* o comportamento é dado pelo elemento **e** pelo *Visitor* sendo utilizado

Visitor

- Implementação:
  - Quem é responsável pela varredura ?
    - Ou “como o *Visitor* chega até o elemento ?”
    - Três opções:
      - O próprio agregado *(collection* ou *composite)* realiza a varredura
      - Utilizar um *Iterator* (interno ou externo) para visitar os elementos
      - O próprio *Visitor* realiza a varredura, embora provavelmente isso irá gerar uma duplicação de código nos *ConcreteVisitor*.
        - A principal motivação é criar varreduras complexas que dependem dos resultados das operações realizadas nos elementos do agregado

Visitor

- Código exemplo:

```
class Equipment {
public:
        virtual ~Equipment();

        const char* Name() { return _name; }

        virtual Watt Power();
        virtual Currency NetPrice();
        virtual Currency DiscountPrice();

        virtual void Accept(EquipmentVisitor&);
protected:
        Equipment(const char*);
private:
        const char* _name;
};
```

```
void FloppyDisk::Accept (EquipmentVisitor& visitor) {
        visitor.VisitFloppyDisk(this);
}
```

Visitor

- Código exemplo:

```
void Chassis::Accept (EquipmentVisitor& visitor) {
        for (
                ListIterator i(_parts);
                !i.IsDone();
                i.Next()
        ) {
                i.CurrentItem()->Accept(visitor);
        }
        visitor.VisitChassis(this);
}
```

Visitor

- Código exemplo:

```
class EquipmentVisitor {
public:
        virtual ~EquipmentVisitor();

        virtual void VisitFloppyDisk(FloppyDisk*);
        virtual void VisitCard(Card*);
        virtual void VisitChassis(Chassis*);
        virtual void VisitBus(Bus*);

        // and so on for other concrete subclasses of Equipment
protected:
        EquipmentVisitor();
};
```

Visitor

- Código exemplo:

```
class PricingVisitor : public EquipmentVisitor {
public:
        PricingVisitor();

        Currency& GetTotalPrice();

        virtual void VisitFloppyDisk(FloppyDisk*);
        virtual void VisitCard(Card*);
        virtual void VisitChassis(Chassis*);
        virtual void VisitBus(Bus*);
        // ...
private:
        Currency _total;
};
```

```
void PricingVisitor::VisitFloppyDisk (FloppyDisk* e) {
        _total += e->NetPrice();
}

void PricingVisitor::VisitChassis (Chassis* e) {
        _total += e->DiscountPrice();
}
```

Visitor

- Código exemplo:

```
class InventoryVisitor : public EquipmentVisitor {
public:
        InventoryVisitor();

        Inventory& GetInventory();

        virtual void VisitFloppyDisk(FloppyDisk*);
        virtual void VisitCard(Card*);
        virtual void VisitChassis(Chassis*);
        virtual void VisitBus(Bus*);
        // ...
private:
        Inventory _inventory;
};
```

```
void InventoryVisitor::VisitFloppyDisk (FloppyDisk* e) {
        _inventory.Accumulate(e);
}

void InventoryVisitor::VisitChassis (Chassis* e) {
        _inventory.Accumulate(e);
}
```

Visitor

- Código exemplo:

```
Equipment* component;
InventoryVisitor visitor;

component->Accept(visitor);
cout << "Inventory "
          << component->Name()
          << visitor.GetInventory();
```

Visitor

- Usos conhecidos:
  - *Smalltalk*
  - *IRIS Inventor*

Visitor

- Padrões relacionados:
  - *Visitors* podem ser utilizados para aplicar uma operação em um agregado implementado sob a forma de um *Composite*
  - O *Visitor* pode ser utilizado para implementa a interpretação, quando utilizando o *Interpreter*

INF011 – Padrões de Projeto

26 – *Visitor*

Sandro Santos Andrade
sandroandrade@ifba.edu.br

Instituto Federal de Educação, Ciência e Tecnologia da Bahia
Departamento de Tecnologia Eletro-Eletrônica
Graduação Tecnológica em Análise e Desenvolvimento de Sistemas