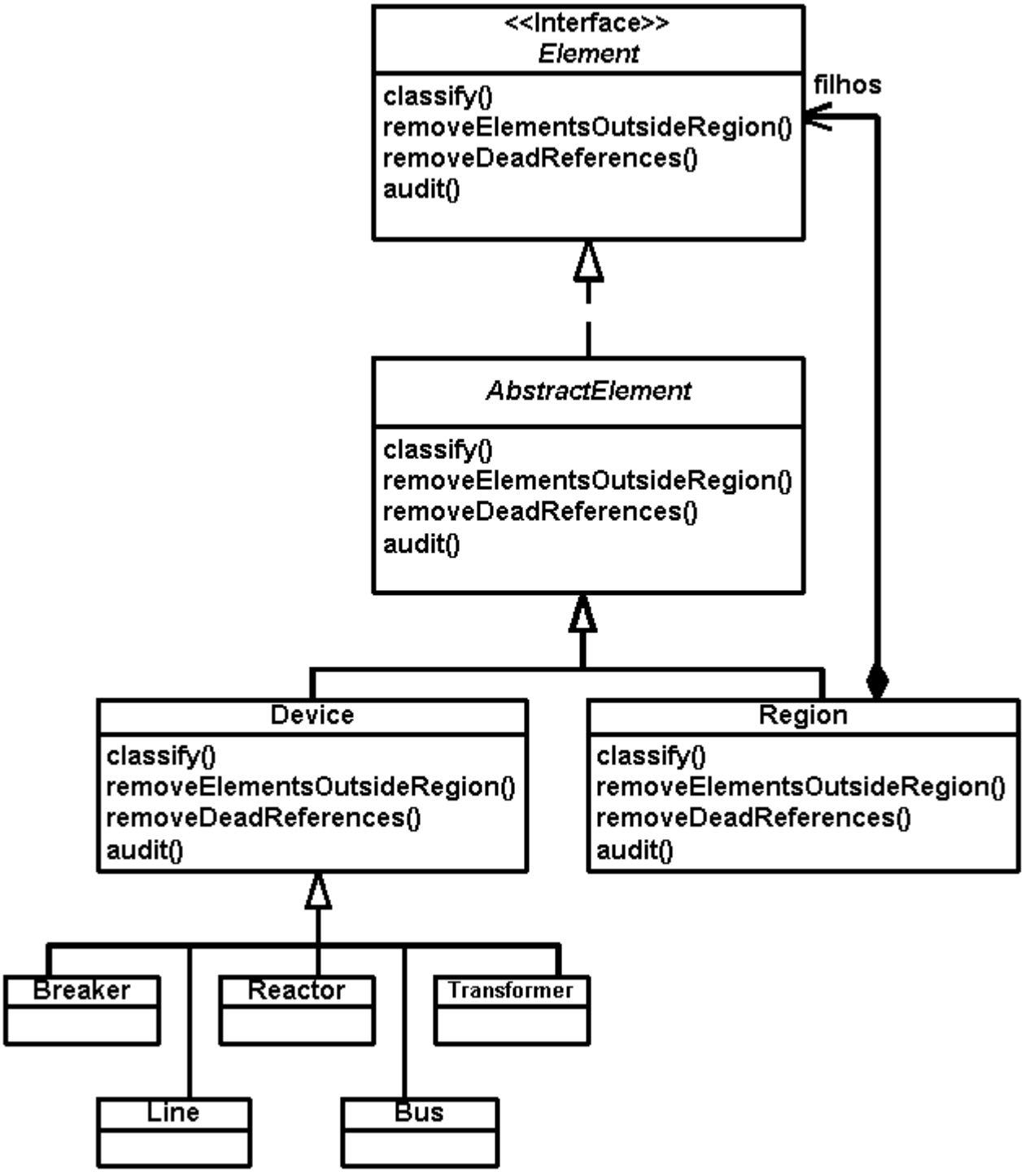
Visitor

Um problema a resolver

- Temos uma hierarquia de classes, provavelmente um Composite
- Exemplo: Numa rede elétrica, temos a seguinte hierarquia:



- Esta hierarquia está sendo usada num programa que analisa uma rede elétrica e manipula seus elementos de várias formas
  - O processamento do Composite tem várias fases
    - Primeira fase: classificar os elementos (Central Breaker, Line Breaker, Line Reactor, Bus reactor, etc.)
      - A classificação depende da topologia (o que está a redor de cada elemento, seus vizinhos, etc.)
      - Nesta fase, aproveita-se para criar associações entre elementos relacionados (ex. o reator e a linha do qual ele é reator)
    - Segunda fase: remover certos elementos desinteressantes, por estarem fora da região de interesse
      - Um elemento desinteressante é marcado como tal
    - Terceira fase: remover as referências que elementos têm a outros elementos desinteressantes
    - Quarta fase: Auditar o resultado do trabalho acima, para ver se não faltou nada
- Os métodos que precisamos ter, para cada elemento da hierarquia são:
  - classify()
  - removeElementsOutsideRegion()
  - removeDeadReferences()
  - audit()
- Observe que a implementação de cada método depende da classe do elemento
  - Essas operações são todas polimórficas
  - Exemplo: a operação classify() será diferente para um Breaker, uma Line, um Reactor, etc.
- É importante notar que essas fases foram aparecendo durante o desenvolvimento do programa

- É altamente provável que apareçam mais fases no futuro
- Precisaremos adicionar novos métodos (estender) os elementos da hierarquia

- A solução de acrescentar métodos às classes da hierarquia é ruim:
  - Espalha o código (ex. a auditoria está espalhada em muitas classes)
  - Cada extensão necessita mexer em muitas classes e recompilá-las
- Uma solução usa o padrão Visitor
  - Visitor encapsula as extensões de funcionalidade em objetos separados

O padrão Visitor

Objetivo

- Representar uma operação a ser realizada nos elementos de uma estrutura de objetos
- Visitor permite que você defina uma nova operação sem alterar as classes dos elementos nos quais a operação atua

Resumo

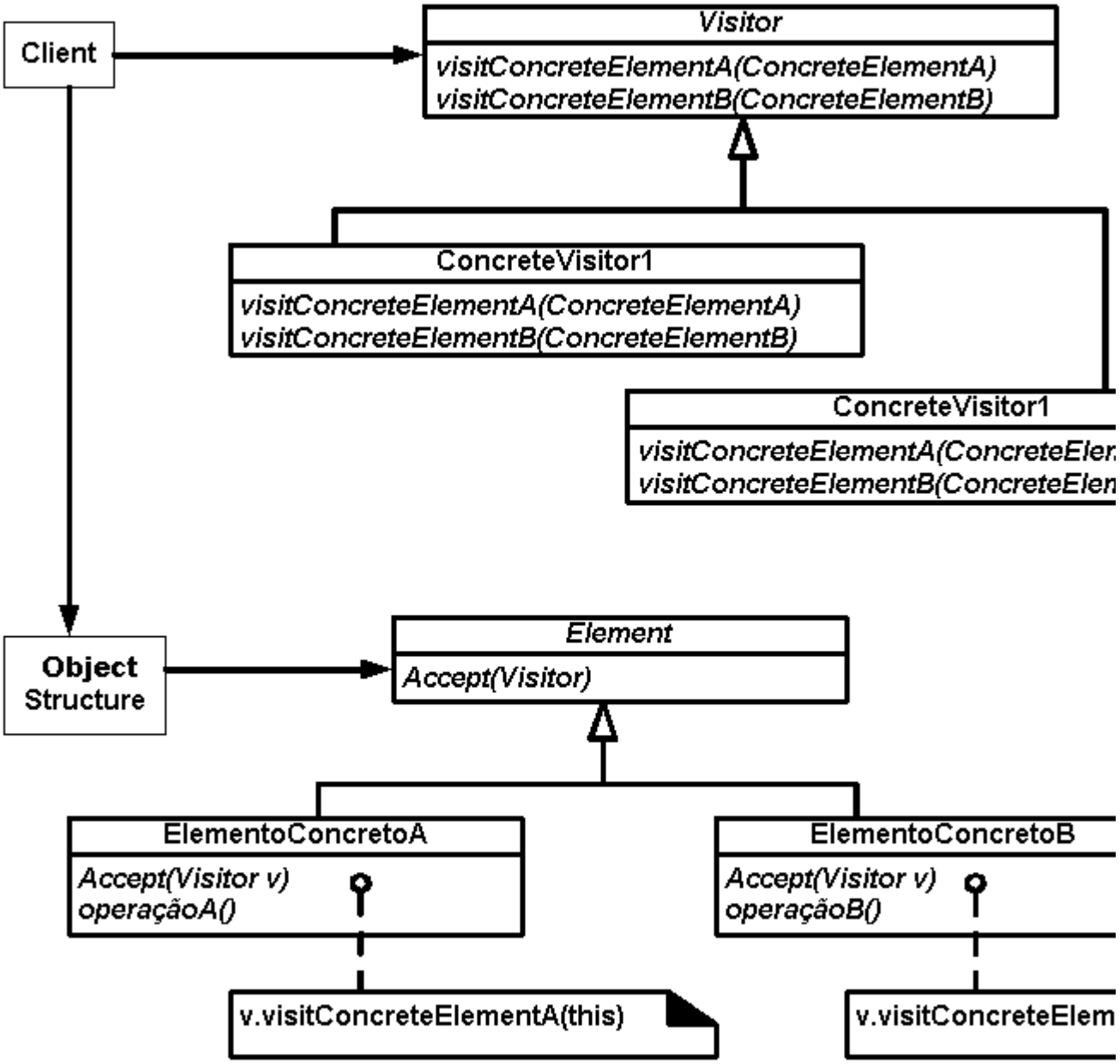
- Faça com que cada elemento aceite um objeto "visitante" e o chame através de um método apropriado
- Para cada nova operação que se deseja acrescentar, cria-se um objeto Visitante novo
  - Exemplo
    - Para auditar, crie um objeto da classe AuditVisitor
    - Este objeto tem os métodos visitBreaker (Breaker), visitLine(Line), etc.
- Cada elemento da hierarquia aceita o visitante com o método accept(Visitor) e chama o método apropriado

do vistante
- Em resumo: a operação a ser feita dependerá tanto do tipo de Visitor quanto do tipo de elemento que o Visitor visite

Quando usar o padrão Visitor?

- Uma estrutura de objetos contém muitas classes de objetos com interfaces diferentes e você deseja realizar operações nestes objetos que dependem das classes concretas
- Muitas operações distintas e não relacionadas devem ser realizadas numa estrutura de objetos e você quer evitar "poluir" as classes com estas operações
  - Visitor permite que você agrupe as operações relacionadas numa mesma classe (digamos AuditVisitor)
- As classes que definem a estrutura de objetos raramente muda mas a adição de novas operações é freqüente.
  - Se as operações fossem colocadas na classes da estrutura, a mudança seria mais complexa
  - Porém, se as classes da estrutura de objetos mudar com freqüência, é melhor deixar as operações junta às classes da estrutura

Estrutura genérica



Participantes

- **Visitor** (AbstractVisitor)
  - Declara uma operação de visita para cada classe de ElementoConcreto na estrutura de objetos
    - O nome da operação identifica a classe que está chamando o Visitor (ex. visitBreaker())
    - Desta forma, o Visitor sabe a classe concreta do elemento sendo visitado e pode acessar este objeto por sua interface
- **ConcreteVisitor** (AuditVisitor)
  - Implementa cada operação declarada pelo Visitor.

Cada operação implementa uma fragmento do algoritmo definido para a classe correspondente de objetos na estrutura
    - O ConcreteVisitor pode acumular estado durante a varredura da estrutura de objetos
- **Element** (Element)
    - Define uma operação accept() que recebe um Visitor
- **ConcreteElement** (Breaker)
    - Implementa uma operação accept() que recebe um Visitor e chama a operação de visita apropriada deste Visitor
        - Exemplo: visitBreaker()
- **Object Structure**
    - Pode enumerar seus elementos
    - Pode prover uma interface de mais alto nível para que o Visitor visite os elementos
    - É freqüentemente um Composite mas pode ser uma coleção simples

Colaborações

- O cliente deve criar um Visitor e visitar todos os elementos com este Visitor
- A estrutura de objetos (ou o cliente) chama visit (Visitor) para cada elemento e cada elemento chama visitConcreteElementXpto(this) do Visitor
    - Note que o Visitor recebe o elemento sendo visitado para poder acessá-lo se desejado

Conseqüências do padrão Visitor

- Visitor permite adicionar novas operações com facilidade
    - Basta adicionar um novo Visitor
    - Se a funcionalidade estivesse espalhada nas classes da estrutura, cada classe deveria ser alterada para introduzir a nova operação

- Um Visitor junta operações relacionadas e separa operação não relacionadas
  - Comportamento relacionado não está espalhado (coesão), mas localizado num Visitor
  - Comportamento não relacionado está presente em objetos Visitor diferentes
- Adicionar novas classes ConcreteElement é difícil
  - Cada ConcreteVisitor tem que ser mudado para adicionar uma operação para visitar este tipo de elemento
  - Use Visitor se as classes da estrutura são estáveis, mas operações são freqüentemente adicionadas
- Visitas em hierarquias diferentes
  - Observe que um Visitor pode muito bem definir operações de visitas em elementos que não participem da mesma hierarquia
  - Isso não seria possível se, por exemplo, um mero iterador fosse usado para varrer a estrutura
- Acumulação de estado
  - Um Visitor pode acumular estado durante as visitas
  - Sem este objeto Visitor, o estado deveria ser acumulado num objeto adicional passado como parâmetro (padrão "Collecting Parameter")
- Quebra de encapsulação
  - Já que a operação está dentro de um Visitor, e fora de um ConcreteElement, o ConcreteElement poderá ter que expor mais interfaces do que seria desejável para que o Visitor possa acessá-lo
  - Isso pode comprometer a encapsulação

Considerações de implementação

- Veremos um exemplo de código adiante
- Double Dispatch
  - Em linguagens comuns (C++, Smalltalk, Java), o método chamado depende da mensagem e do objeto receptor

- Algumas linguagens, tais como CLOS (mas não C++, Smalltalk, Java) oferecem Double Dispatch, em que o método chamado depende da mensagem e de *dois* outros objetos
- Visitor essencialmente simula Double Dispatch, pois a operação realizada depende do tipo do Visitor e do tipo elemento visitado
  - Em vez de usar binding dinâmico simples colocando a operação na interface do Elemento, junatm-se as operações num Visitor e usamos accept() para realizar um nível a mais de dispatch em tempo de execução

- Quem é responsável pode varrer a estrutura de objetos?
  - Um Visitor deve visitar todos os elementos de uma estrutura
  - Quem tem a responsabilidade de realizar a varredura da estrutura e chamar accept() em cada elemento?
    - A iteração pode ser feita na estrutura de objetos
      - Itera-se sobre os elementos, chamando accept() para cada um
      - Num Composite, o accept() de um elemento composto pode chamar accept() dos filhos
    - Podemos usar um iterador, fora da estrutura de objetos
      - Se o iterador for externo, é o cliente que faz a varredura
      - Se o iterador for interno, é o próprio iterador que chama accept()
        - Equivalente a fazer a estrutura se responsável pela iteração
        - Porém, neste caso, o iterador pode chamar o Visitor diretamente, em cujo caso não haverá double dispatch
    - A iteração pode ser feita no próprio Visitor, mas, neste caso, cada Visitor vai ter que repetir o código de varredura

- Às vezes é necessário fazer isso se a varredura depender do resultado das operações realizadas pelo Visitor

Exemplo de código

- A estrutura de objetos segue

```
ublic interface Element extends Comparable {
  // composite design pattern being used

  public final int TYPE_UNDEFINED = 0;
  public final int TYPE_BREAKER = 1;
  public final int TYPE_RECLOSER = 2;
  public final int TYPE_SWITCH = 3;
  public final int TYPE_SUBSTATION = 4;
  public final int TYPE_TRANSFORMER = 5;
  public final int TYPE_LINE = 6;
  public final int TYPE_STATIC_COMPENSATOR = 7;
  public final int TYPE_GENERATOR = 8;
  public final int TYPE_SYNCHRONOUS_COMPENSATOR = 9;
  public final int TYPE_BUS = 10;
  public final int TYPE_PHANTOM_BUS = 11;
  public final int TYPE_REACTOR = 12;
  public final int TYPE_CAPACITOR_BANK = 13;
  public final int TYPE_REGION = 14;

  public final int SUBTYPE_UNDEFINED = 0;
  public final int SUBTYPE_BREAKER_REACTOR = 1;
  public final int SUBTYPE_BREAKER_LINE = 2;
  public final int SUBTYPE_BREAKER_TRANSFORMER = 3;
  public final int SUBTYPE_BREAKER_CENTRAL = 4;
  public final int SUBTYPE_BREAKER_TRANSFER_1 = 5;
  public final int SUBTYPE_BREAKER_TRANSFER_2 = 6;
  public final int SUBTYPE_BREAKER_SYNCHRONOUS_COMPENSATOR =
  public final int SUBTYPE_BREAKER_CAPACITOR_BANK = 8;
  public final int SUBTYPE_REGION_SYSTEM = 9;
  public final int SUBTYPE_REGION_REGIONAL_CENTER = 10;
  public final int SUBTYPE_BUS_1 = 11;
  public final int SUBTYPE_BUS_2 = 12;

  // ...

  public void accept(Visitor visitor) throws Exception;
  // ...
}
```

```
public class Region extends AbstractElement {
  // ...
  public void accept(Visitor visitor) throws Exception {
    visitor.visitRegion(this);
    Iterator it = getSortedSubRegionsIterator();
    while(it.hasNext()) {
      Element subregion = (Element)it.next();
      subregion.accept(visitor);
    }
  }
}

public class Device extends ElementTwoNodes {
  // ...
  public void accept(Visitor visitor) throws Exception {
    visitor.visitDevice(this);
  }
}

public class Breaker extends Device {
  // ...
  public void accept(Visitor visitor) throws Exception {
    visitor.visitBreaker(this);
  }
}
// ...
```

- O código principal segue

```
public class Convert {
  private static Convert instance = null;
  private Properties substationNameToCentersMap;
  private Map allElements = new HashMap();
  private Map substationIdToCentersMap = new HashMap();

  public static void main(String[] args) {
    configLog();
    if(args.length != 2) {
      syntax();
    }
    try {
      Region theWholeThing = Convert.getInstance().solve(arg
      System.out.println(theWholeThing.toXml());
    } catch (Exception ex) {
      logger.error("Excecao: " + ex.toString());
    }
```

```
  }

  private Region solve(String dataFileName, String substatio
        throws Exception {
    // create an object structure
    Region theWholeThing = load(dataFileName, substationFile
    classify(theWholeThing);
    removeLinesOutsideARegion(theWholeThing);
    removeLineReferences(theWholeThing);
    audit(theWholeThing);
    return theWholeThing;
  }

  private void classify(Region topRegion) throws Exception {
    topRegion.accept(new ClassifierVisitor());
  }

  public void removeLinesOutsideARegion(Region topRegion) th
    topRegion.accept(new RemoveElementsOutsideARegionVisitor
  }

  public void removeLineReferences(Region topRegion) throws
    topRegion.accept(new RemoveLineReferencesVisitor());
  }

  private void audit(Region topRegion) throws Exception {
    topRegion.accept(new AuditVisitor());
  }

  // ...
}
```

- A estrutura dos Visitors segue

```
public interface Visitor {
  public void visitBreaker(Element element) throws Exception
  public void visitDevice(Element element) throws Exception;
  public void visitRegion(Element element) throws Exception;
  public void visitReactor(Element element) throws Exception
  public void visitLine(Element element) throws Exception;
}

public abstract class AbstractVisitor implements Visitor {
  public void visitLine(Element element) throws Exception {}
  public void visitReactor(Element element) throws Exception
  public void visitDevice(Element element) throws Exception
  public void visitRegion(Element element) throws Exception
```

```
  public void visitBreaker(Element element) throws Exception
}

public class AuditVisitor extends AbstractVisitor {
  static Logger logger = Logger.getLogger(AuditVisitor.class

  public void visitBreaker(Element element) {
    if(!element.acceptableElement()) {
      return;
    }
    if(element.getAssociatedLine() == null &&
       element.getAssociatedTransformer() == null) {
      logger.info(element.getTypeName() + " " + element.getI
                  " nao foi associado a uma regra de linha/t
    }
  }

  public void visitReactor(Element element) {
    if(!element.acceptableElement()) {
      return;
    }
    if(element.getAssociatedLine() == null) {
      logger.info(element.getTypeName() + " " + element.getI
                  " nao foi associado a uma regra de linha")
    }
  }
  public void visitLine(Element element) {
    if(!element.acceptableElement()) {
      return;
    }
    if(element.getAssociatedBreaker1() == null) {
      logger.info(element.getTypeName() + " " + element.getI
                  " nao tem disjuntor de linha");
    }
  }
}

// etc. para outros Visitors
```

Pergunta final para discussão

- Como escrever Visitors para ConcreteElements que participam de uma hierarquia (ex. OilBreaker é um tipo de Breaker)? Quando OilBreaker chama visitOilBreaker(), o que deve accontecer, já que OilBreaker também é um Breaker?

programa