INF011 – Padrões de Projeto

11 – *Composite*

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

Composite

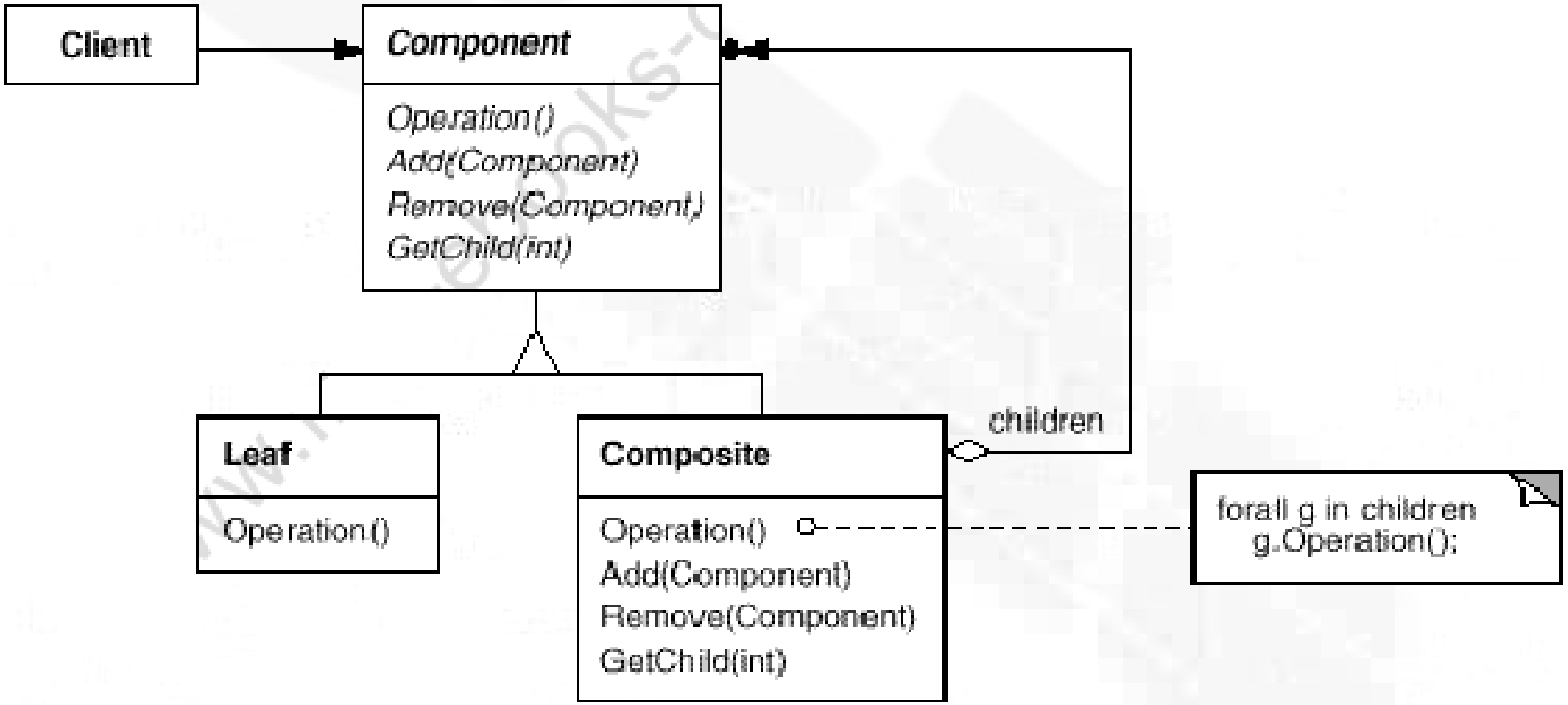
- Propósito:
  - Compor objetos em uma estrutura de árvore com o objetivo de representar hierarquias do tipo parte-todo
- Motivação:
  - Editor gráfico que permite que o usuário crie diagramas complexos a partir de componentes mais simples
  - Componentes podem ser agrupados para formar componentes maiores que também podem ser agrupados
  - Editor gráfico = primitivas gráficas (reta, círculo, texto, etc) + *containers* para estas primitivas
  - Embora o usuário trate primitivas e *containers* da mesma forma precisa-se diferenciá-los no código-fonte

Composite

- Aplicabilidade:
  - Deseja-se representar hierarquias do tipo todo-parte
  - Deseja-se que clientes possam ignorar as diferenças entre composições de objetos e objetos individuais
    - Clientes irão tratar uniformemente todos os objetos pertencentes a uma determinada estrutura de composição

Composite

- Estrutura:



Composite

- Participantes:
  - *Component* (Graphics):
    - Declara a interface para objetos da composição
    - Implementa comportamento *default* para a interface comum a todas as classes, conforme apropriado
    - Declara a interface para acessar e gerenciar os componentes-filho
    - (opcional) define uma interface para acessar o pai de um componente da estrutura recursiva. Se apropriado pode também implementar esta interface
  - *Leaf* (Rectangle, Line, Text, etc):
    - Representa objetos-folha na composição (aqueles que não possuem filhos)
    - Define o comportamento dos objetos primitivos

Composite

- Participantes:
  - *Composite* (Picture):
    - Define o comportamento dos componentes que possuem filhos
    - Armazena os componentes-filho
    - Implementa as operações relacionadas a filhos presentes na interface do *Component*
  - *Client*:
    - Manipula objetos da composição através da interface *Component*

Composite

- Colaborações:
  - Os clientes utilizam a interface *Component* para interagir com objetos da estrutura composta:
    - Se o receptor da ação for uma folha a requisição é atendida imediatamente
    - Se o receptor da ação for um *composite* a requisição é repassada para os seus componentes-filho, possivelmente realizando operações adicionais antes ou depois do repasse

Composite

- Consequências:
  - Define hierarquias de classes formadas por objetos primitivos e objetos *composite*:
    - Todo código de cliente que espera um objeto primitivo podem também manipular um objeto *composite*
  - Torna o cliente simples:
    - Clientes tratam estruturas compostas e objetos individuais de maneira uniforme
    - Clientes não sabem se eles estão lidando com um objeto *composite* ou com um objeto primitivo
    - Evita sentenças *switch-case* sobre as classes que definem a composição

Composite

- Consequências:
  - Torna fácil a adição de novos tipos de componentes:
    -
      - Sub-classes de *Composite* ou de qualquer classe-folha funcionarãos

Composite

- Implementação:
  - Referências explícitas para o pai:
    - Podem simplificar a navegação e gerenciamento da estrutura composta
    - Simplifica as operações de remoção de um componente e movimentação para cima na estrutura
    - Ajudam na implementação do padrão *Chain of Responsibility*
    - Esta referência é geralmente armazenada na classe *Component*
    - Portanto as classes-folha e *Composite* herdam esta referência e as operações que a manipulam
    - Invariante a ser mantida: todos os filhos de um *composite* possuem como pai o *composite* que os possui como filhos

Composite

- Implementação:
  - Compartilhando componentes:
    - É frequentemente útil para reduzir o espaço necessário para armazenamento
    - É difícil quando componentes devem possuir apenas um único pai
    - Pode-se definir que um filho armazenará múltiplos pais, porém dificulta as operações de movimentação para cima na estrutura
    - Solução: padrão *Flyweight* – evita armazenar múltiplos pais em conjunto através da externalização de parte do estado do filho

Composite

- Implementação:
  - Maximizando a interface de *Component*:
    - Uma das metas do *Composite* é fazer com que os clientes tenham uma visão uniforme de *Leaf* e *Composite*
    - Para isso, a interface *Component* deve definir o máximo possível de operações comuns a *Leaf* e *Composite*, geralmente com implementações *default*
    - Isto pode resultar em operações que não fazem sentido para **todas** as suas sub-classes (ex: acessar os filhos não faz sentido para *Leaves*)
    - Entretando, pode-se definir que um *Leaf* é um componente que nunca tem filhos e esta será a implementação *default*

Composite

- Implementação:
  - Declarando as operações de gerenciamento de filhos:
    - A classe *Composite* **implementa** as operações *add()* e *remove()*, porém onde elas devem ser **declaradas** ?
    - A solução envolve um *trade-off* entre robustez e transparência:
      - Declarando em *Component*: obtém-se transparência *(Leaves* e *Composites* são tratados uniformemente) porém perde-se em robustez (clientes podem adicionar ou remover objetos em *Leaves*)
      - Declarando em *Composite*: obtém-se robustez (qualquer tentativa de adição ou remoção de filhos em *Leaves* irá gerar um erro de compilação) porém perde-se em transparência *(Leaves* e *Composites* terão interfaces diferentes*)*

Composite

- Implementação:
  - Declarando as operações de gerenciamento de filhos:
    - Se optarmos por rubustez como diferenciar *Leaves* de *Composites* ?

```
class Composite;

class Component {
public:
    //...
    virtual Composite* GetComposite() { return 0; }
};

class Composite : public Component {
public:
    void Add(Component*);
    // ...
    virtual Composite* GetComposite() { return this; }
};

class Leaf : public Component {
    // ...
};
```

- Pode-se também utilizar o *dynamic_cast* do C++

Composite

- Implementação:
  - Declarando as operações de gerenciamento de filhos:
    - A única forma de prover transparência é definir implementações *default* de *add()* e *remove()* em *Component*
    - Mas qual seria esta implementação *default* ?
      - Não fazer nada: ignora o fato que tentar adicionar um filho a um *Leaf* provavelmente é um erro de código
      - Uma solução melhor é:
        - Fazer com que *add*() por *default* falhe (por exemplo, gerando uma exceção)
        - Fazer com que *remove()* por *default* falhe se o seu parâmetro não for um filho do componente
      - Outra solução:
        - Se mantivermos referência para o pai, mudamos a semântica de *remove()* (agora sem argumentos) para remover o componente da lista de filhos do pai

Composite

- Implementação:
  - A classe *Component* deve armazenar a lista de filhos ?
    - Geralmente não pois todos os *Leaves* iriam também armazenar esta lista, mesmo sem nunca terem filhos
    - Só é viável quando o número de folhas é relativamente baixo
  - Ordenação dos filhos:
    - No exemplo do editor gráfico poderia refletir o *z-order*
    - No caso de *parse trees* a ordenação dos filhos deve estar de acordo com o programa representado
    - Quando a ordenação é necessária as interfaces de acesso e gerenciamento de filhos devem ser cuidadosamente projetadas. O padrão *Iterator* pode ser útil

Composite

- Implementação:
  - Melhorando o desempenho com *caching*:
    - Se precisa-se percorrer ou realizar buscas em uma composição a classe *Composite* pode realizar *caching* das informações sobre os filhos
    - Pode-se armazenar as informações reais dos filhos ou informações que permitem otimizar o processo de varredura ou busca
      - Ex: Pode-se fazer *caching* do *bounding box* de cada filho no exemplo do editor gráfico. Durante a exibição ou seleção de um componente o editor poderia abortar aqueles *composites* cujo *bounding box* está fora da região visível da tela
      - Precisa-se, entretanto, de mecanismos para invalidação dos *caches* do pai quando algum filho for modificado. Ex: referências explícitas para o pai e interfaces para invalidação

Composite

- Implementação:
  - Quando remover os componentes ?
    - Em linguagens sem *garbage collection* é geralmente melhor fazer com que o *composite* se responsabilize em remover os seus filhos que ele for destruído
    - Exceção: quando objetos *Leaf* forem *read-only* (imutáveis) e compartilhados
      - Solução: *reference-counting* ?
  - Qual a melhor estrutura de dados para armazenar os componentes ?
    - Depende da eficiência desejada (listas encadeadas, árvores, *arrays*, tabelas *hash*)

Composite

- Código exemplo (*Component*)*:*

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

    const char* Name() { return _name; }

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

    virtual void Add(Equipment*);
    virtual void Remove(Equipment*);
    virtual Iterator* CreateIterator();
protected:
    Equipment(const char*);
private:
    const char* _name;
};
```

Composite

- Código exemplo (*Leaf*):

```
class FloppyDisk : public Equipment {
public:
    FloppyDisk(const char*);
    virtual ~FloppyDisk();

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

Composite

- Código exemplo (*Composite*):

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

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

    virtual void Add(Equipment*);
    virtual void Remove(Equipment*);
    virtual Iterator* CreateIterator();

protected:
    CompositeEquipment(const char*);
private:
    List _equipment;
};
```

Composite

- Código exemplo (*Composite*):

```
Currency CompositeEquipment::NetPrice () {
    Iterator* i = CreateIterator();
    Currency total = 0;

    for (i->First(); !i->IsDone(); i->Next()) {
        total += i->CurrentItem()->NetPrice();
    }
    delete i;
    return total;
}
```

Composite

- Código exemplo (*Composite* específico):

```
class Chassis : public CompositeEquipment {
public:
    Chassis(const char*);
    virtual ~Chassis();

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

Composite

- Código exemplo (*main*):

```
Cabinet* cabinet = new Cabinet("PC Cabinet");
Chassis* chassis = new Chassis("PC Chassis");

cabinet->Add(chassis);

Bus* bus = new Bus("MCA Bus");
bus->Add(new Card("16Mbs Token Ring"));
chassis->Add(bus);
chassis->Add(new FloppyDisk("3.5in Floppy"));

cout << "The net price is " << chassis->NetPrice() << endl;
```

Composite

- Usos conhecidos:
  - Classe *View* do MVC do *Smalltalk*
  - ET++, *InterViews*, *Graphics*, *Glyphs*
  - *Framework* de compilação RTL do *Smalltalk* (*parse trees*)

Composite

- Padrões relacionados:
  - Frequentemente a referência explícita do componente para o pai é utilizada no padrão *Chain of Responsibility*
  - O *Decorator* é frequentemente utilizado com o *Composite*
  - O *Flyweight* permite a implementação de componentes compartilhados, porém não permite mais o uso de referências explícitas ao pai
  - O *Iterator* pode ser utilizado para percorrer *Composites*
  - O *Visitor* localiza operações e comportamentos que, sem o seu uso, estariam distribuídos entre as classes *Composite* e *Leaf*

INF011 – Padrões de Projeto

11 – *Composite*

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