

INF011 – Padrões de Projeto

24 – *Strategy*

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

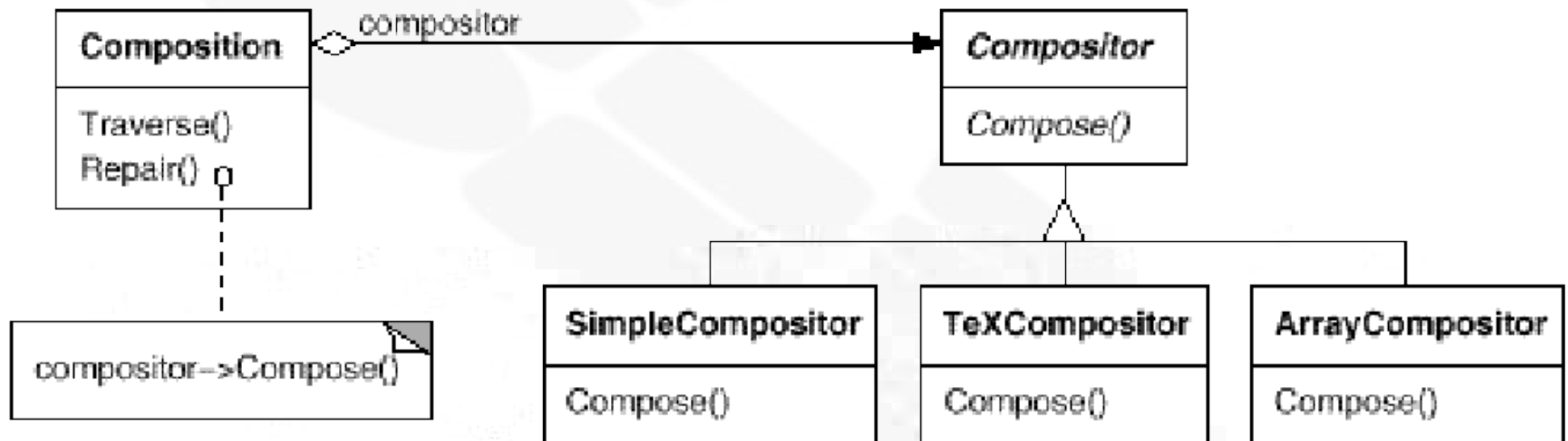


Strategy

- Propósito:
 - Definir uma família de algoritmos, encapsular cada um e torná-los intercambiáveis
- Também conhecido como: *Policy*
- Motivação:
 - Diversos algoritmos para quebra de um texto em linhas
 - Não é desejável acoplar estes algoritmos nos clientes:
 - Se tornam maiores e mais difíceis de manter
 - Diferentes algoritmos serão apropriados em momentos diferentes. Não queremos suportar o que não usaremos
 - É difícil adicionar novos algoritmos ou modificar os já existentes

Strategy

- Motivação:



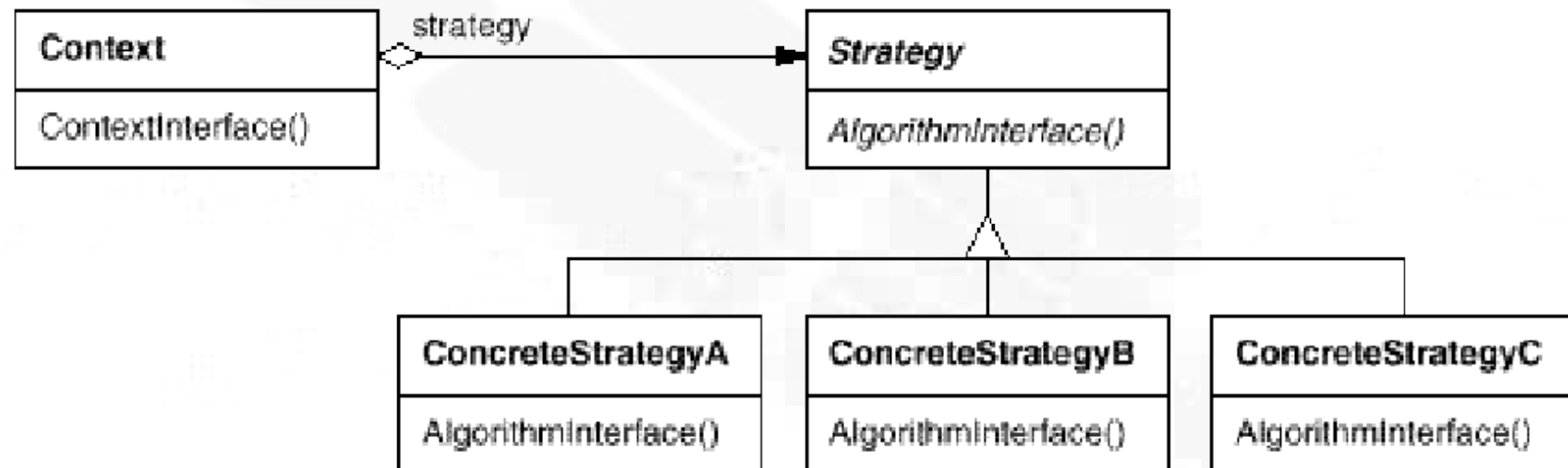
- *SimpleCompositor*: analisa uma linha por vez
- *TeXCompositor*: analisa um parágrafo por vez
- *ArrayCompositor*: número fixo de elementos por linha

Strategy

- Aplicabilidade:
 - Quando muitas classes relacionadas diferem somente no seu comportamento
 - Quando é necessário utilizar diferentes variações de um algoritmo para, por exemplo, refletir *trade-offs* de tempo/espço
 - Quando um algoritmo utiliza dados que não devem estar expostos aos clientes
 - Quando uma classe define múltiplos comportamentos através de sentenças condicionais em seus métodos

Strategy

- Estrutura:



Strategy

- Participantes:
 - *Strategy* (Compositor):
 - Declara uma interface comum a todos os algoritmos suportados
 - O *Context* utiliza esta interface para chamar um algoritmo definido por um *ConcreteStrategy*
 - *ConcreteStrategy* (SimpleCompositor, TeXCompositor etc):
 - Implementa o algoritmo utilizando a interface *Strategy*
 - *Context* (Composition):
 - É configurado com um *ConcreteStrategy*
 - Mantém uma referência para o objeto *Strategy*
 - Pode definir uma interface que permite que os *ConcreteStrategies* acessem seus dados

Strategy

- Colaborações:
 - O *Strategy* e o *Context* interagem para implementar o algoritmo escolhido:
 - O *Context* pode passar todos os dados necessários à execução do algoritmo, ou
 - O *Context* passa ele próprio como argumento da operação, permitindo que o *Strategy* consulte os dados necessários
 - Um *Context* repassa requisições do cliente para o seu *Strategy*. O cliente geralmente cria e configura o *Context* com um *ConcreteStrategy*
 - A partir daí o cliente interage somente com o *Context*

Strategy

- Consequências:
 - Família de algoritmos relacionados
 - Uma hierarquia de classes *Strategy* define uma família de algoritmos/comportamentos. Herança pode ajudar a fatorar comportamento comum entre os algoritmos
 - Uma alternativa a *sub-classing*
 - Poderia-se derivar o *Contexto* para prover diferentes implementações
 - Entretanto o comportamento estará *hard-coded* no *Context*, suas implementações estarão na mesma classe e não possibilita a mudança do algoritmo em *run-time*

Strategy

- Consequências:
 - *Strategies* eliminam sentenças condicionais (ANTES)

```
void Composition::Repair () {  
    switch (_breakingStrategy) {  
        case SimpleStrategy:  
            ComposeWithSimpleCompositor();  
            break;  
        case TeXStrategy:  
            ComposeWithTeXCompositor();  
            break;  
        // ...  
    }  
    // merge results with existing composition, if necessary  
}
```

Strategy

- Consequências:
 - *Strategies* eliminam sentenças condicionais (DEPOIS)

```
void Composition::Repair () {  
    _compositor->Compose();  
    // merge results with existing composition, if necessary  
}
```

Strategy

- Consequências:
 - Escolha entre implementações:
 - Os clientes escolhem entre diferentes estratégias com diferentes complexidades de tempo/espço
 - Clientes devem estar cientes dos diferentes *Strategies*
 - O cliente seleciona um *Strategy* apropriado
 - Deve ser utilizado somente quando a variação do comportamento é relevante para os clientes
 - *Overhead* de comunicação entre o *Strategy* e o *Context*
 - A interface *Strategy* é compartilhada por todos os *ConcreteStrategy*, sejam eles triviais ou complexos
 - O *Context* pode criar e inicializar parâmetros que nunca serão utilizados

Strategy

- Consequências:
 - Número maior de objetos:
 - Pode-se reduzir este *overhead* implementando os *Strategies* como objetos *stateless* e, portanto, compartilháveis
 - Qualquer estado residual é mantido pelo *Context* e passado para o *Strategy* durante a invocação (*Flyweight*)

Strategy

- Implementação:
 - Definindo as interfaces *Strategy* e *Context*
 - Tais interfaces devem fornecer a um *ConcreteStrategy* acesso eficiente aos dados necessários do *Context* e vice-versa
 - Os mesmos modelos *Pull* e *Push* do *Observer* são aqui aplicados

Strategy

- Implementação:
 - *Strategies* como parâmetros *template*

```
template <class AStrategy>
class Context {
    void Operation() { theStrategy.DoAlgorithm(); }
    // ...

private:
    AStrategy theStrategy;
};

class MyStrategy {
public:
    void DoAlgorithm();
};

Context<MyStrategy> aContext;
```

Strategy

- Implementação:
 - Tornando o objeto *Strategy* opcional
 - O *Context* pode ser simplificado se for útil não utilizar um objeto *Strategy*
 - Se existe um objeto *Strategy* ele será utilizado, caso contrário será executado um comportamento *default*
 - Com isso, clientes satisfeitos com o comportamento *default* não precisam se preocupar em configurar o *Context* com um *Strategy*

Strategy

- Código exemplo:

```
class Composition {
public:
    Composition(Compositor*);
    void Repair();
private:
    Compositor* _compositor;
    Component* _components;    // the list of components
    int _componentCount;      // the number of components
    int _lineWidth;           // the Composition's line width
    int* _lineBreaks;         // the position of linebreaks
                                // in components
    int _lineCount;           // the number of lines
};
```

Strategy

- Código exemplo:

```
class Compositor {  
public:  
    virtual int Compose(  
        Coord natural[], Coord stretch[], Coord shrink[],  
        int componentCount, int lineWidth, int breaks[]  
    ) = 0;  
protected:  
    Compositor();  
};
```

Strategy

- Código exemplo:

```
void Composition::Repair () {  
    Coord* natural;  
    Coord* stretchability;  
    Coord* shrinkability;  
    int componentCount;  
    int* breaks;  
  
    // prepare the arrays with the desired component sizes  
    // ...  
  
    // determine where the breaks are:  
    int breakCount;  
    breakCount = _compositor->Compose(  
        natural, stretchability, shrinkability,  
        componentCount, _lineWidth, breaks  
    );  
  
    // lay out components according to breaks  
    // ...  
}
```

Strategy

- Código exemplo:

```
class SimpleCompositor : public Compositor {  
public:  
    SimpleCompositor();  
    virtual int Compose(  
        Coord natural[], Coord stretch[], Coord shrink[],  
        int componentCount, int lineWidth, int breaks[]  
    );  
    // ...  
};
```

Strategy

- Código exemplo:

```
class TeXCompositor : public Compositor {
public:
    TeXCompositor();
    virtual int Compose(
        Coord natural[], Coord stretch[], Coord shrink[],
        int componentCount, int lineWidth, int breaks[]
    );
    // ...
};
```

Strategy

- Código exemplo:

```
class ArrayCompositor : public Compositor {
public:
    ArrayCompositor(int interval);
    virtual int Compose(
        Coord natural[], Coord stretch[], Coord shrink[],
        int componentCount, int lineWidth, int breaks[]
    );
    // ...
};
```

```
Composition* quick = new Composition(new SimpleCompositor);
Composition* slick = new Composition(new TeXCompositor);
Composition* iconic = new Composition(new ArrayCompositor(100));
```

Strategy

- Usos conhecidos:
 - ET++
 - RTL
 - Booch Components
 - RApp
 - ObjectWindows

Strategy

- Padrões relacionados:
 - *Strategies* compartilhados são geralmente implementados como *Flyweights*

INF011 – Padrões de Projeto

24 – *Strategy*

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

