INF011 – Padrões de Projeto

04 – *Builder*

Sandro Santos Andrade
sandroandrade@ifba.edu.br

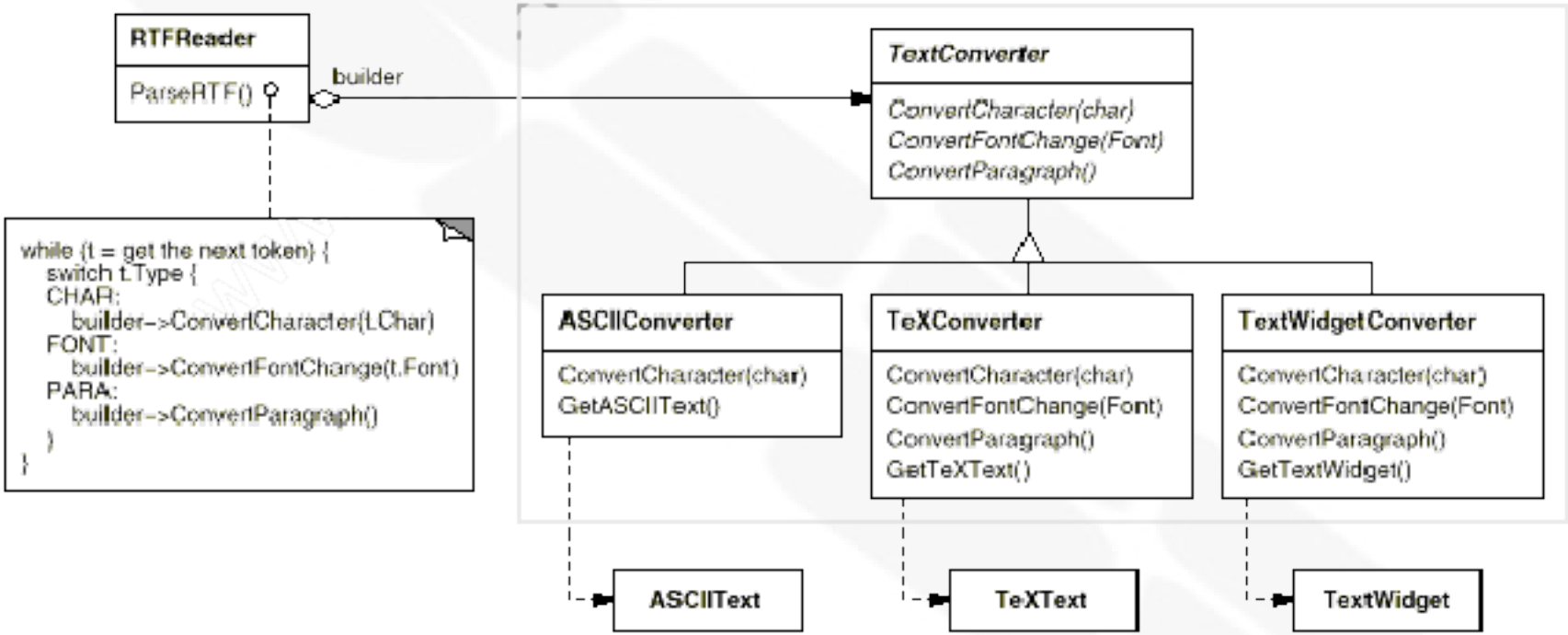
Instituto Federal de Educação, Ciência e Tecnologia da Bahia
Departamento de Tecnologia Eletro-Eletrônica
Graduação Tecnológica em Análise e Desenvolvimento de Sistemas

Builder

- Propósito:
  - Separar a construção de um objeto complexo da sua representação, de modo que o mesmo processo de construção possa criar diferentes representações
- Motivação:
  - Um editor de arquivos RTF (*Rich Text Format*) precisa converter arquivos RTF para uma série de outros formatos
  - O número de possíveis conversões é desconhecido
  - Deve ser fácil acrescentar uma nova conversão sem modificar o editor

Builder

- Motivação:



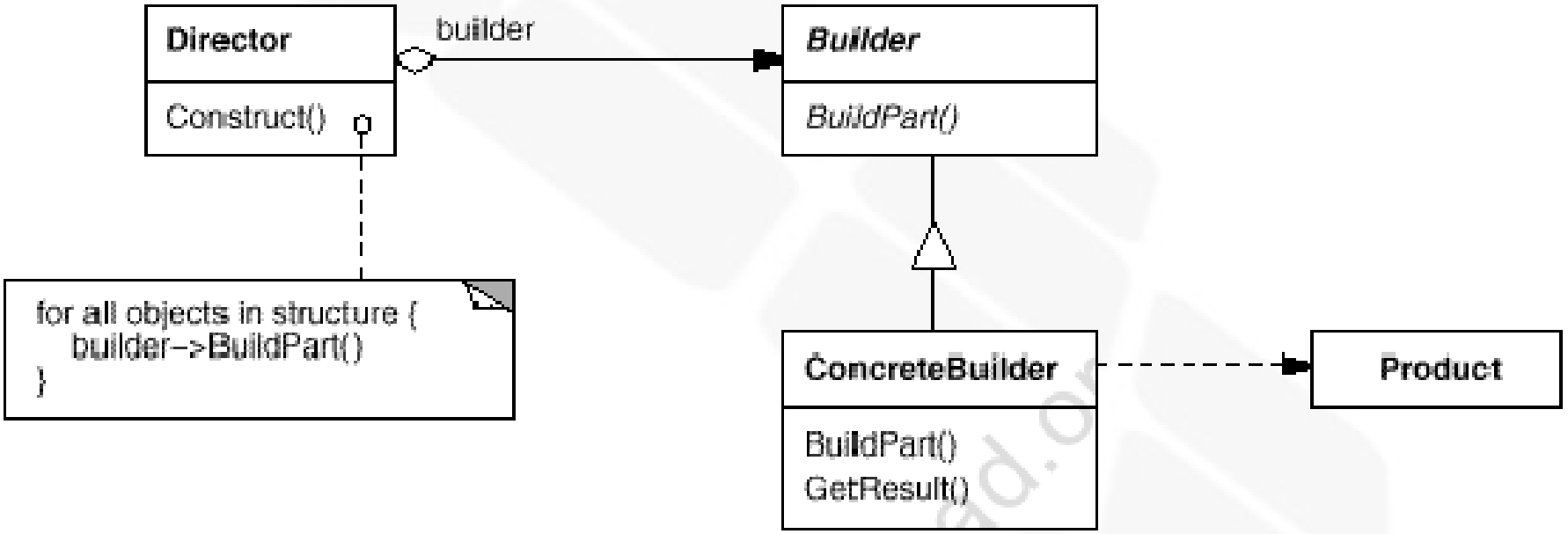
  - Cada conversor implementa os mecanismos para criar e montar o objeto complexo

Builder

- Aplicabilidade:
  - O algoritmo para criar o objeto complexo deve ser independente das partes que compõem o objeto e de como elas são integradas
  - O processo de construção deve permitir diferentes representações do objeto sendo construído

Builder

- Estrutura:



Builder

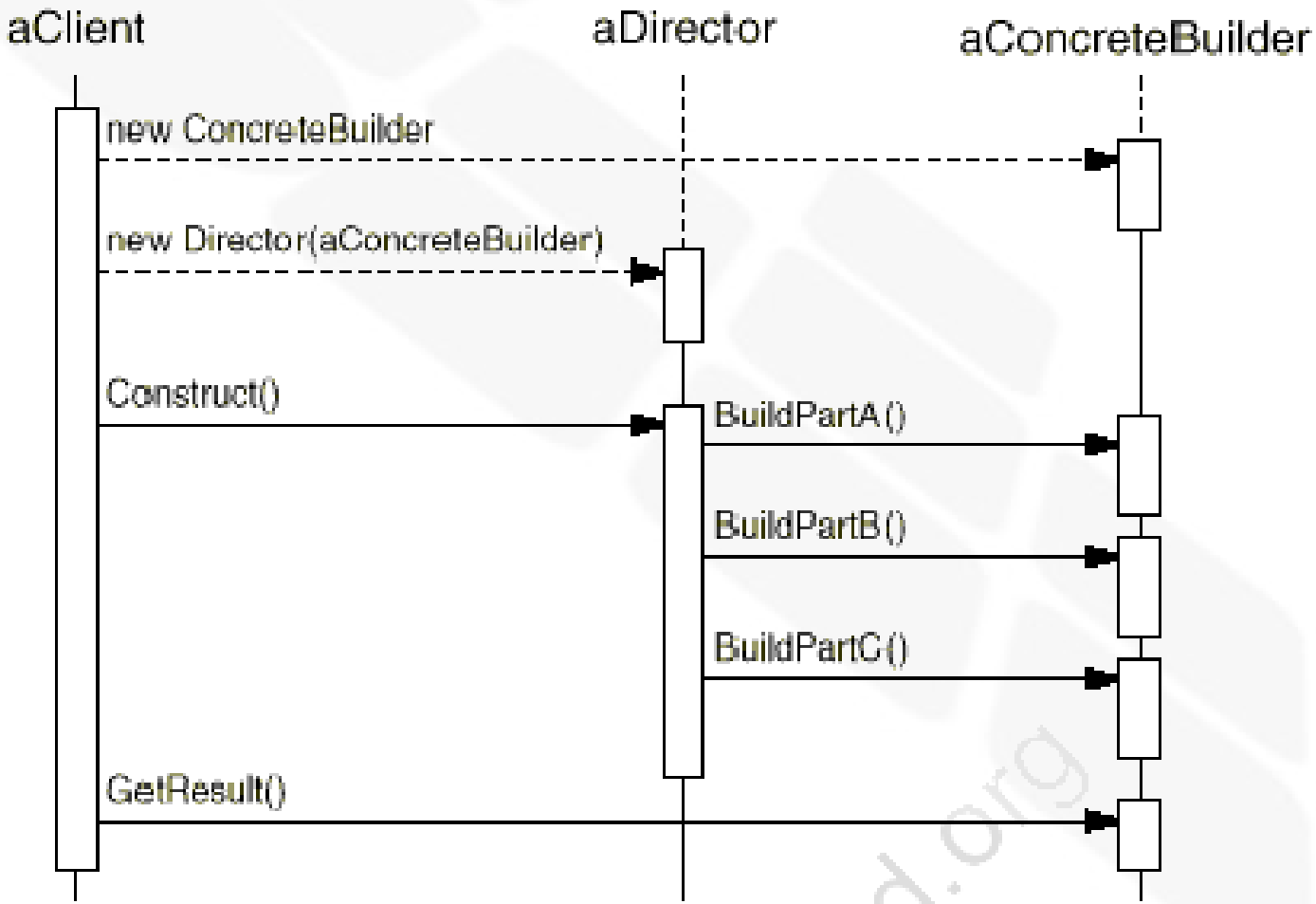
- Participantes:
  - *Builder*: especifica a interface abstrata para a criação das partes do objeto produto
  - *ConcreteBuilder* (ASCIIConverter, TeXConverter, etc):
    - Constrói e monta as partes do objeto através da implementação da interface do *Builder*
    - Define e mantém as representações que ele cria
    - Disponibiliza uma interface para recuperar o produto
  - *Director*: constrói o objeto usando a interface do *Builder*
  - *Product* (ASCIIText, TeXText): representa o objeto complexo sendo construído e inclui classes que definem as partes constituintes

Builder

- Colaborações:
  - O cliente cria o objeto *Director* e o configura com o *Builder* desejado
  - O *Director* notifica o *Builder* sempre que uma parte do produto deve ser construída
  - O *Builder* trata as requisições do *Director* e adiciona partes ao produto
  - O cliente recupera o produto, através do *Builder*

Builder

- Colaborações:



Builder

- Consequências:
  - Permite variar a representação interna do produto:
    - O *Builder* oferece ao *Director* uma interface abstrata de construção do produto que isola a sua estrutura interna e representação, bem como a forma como ele é construido. Para mudar a representação interna do produto basta utilizar outro *Builder* concreto
  - Isola o código de construção e representação do produto:
    - O cliente não precisa conhecer nada sobre as classes que definem a representação interna do produto. Tais classes não aparecem na interface do *Builder*
    - Os mesmos *Builders* concretos podem ser utilizados por outros *Directors* para construir os mesmos produtos a partir de outras representações formadas pelas mesmas partes

Builder

- Consequências:
  - Permite um maior controle do processo de construção do produto:
    - Não constrói o objeto com uma única operação e sim através de múltiplos passos, sob o controle do *Director*
    - Somente ao fim do processo de construção o produto é entregue, pelo *Builder*, ao cliente
    - A interface do *Builder* dá maior ênfase ao processo de construção do objeto, se comparado com outros *creational patterns*

Builder

- Implementação:
  - Interface de construção e montagem:
    - A interface do *Builder* deve ser geral o suficiente para permitir a construção de produtos por todos os possíveis tipos de *Builders*
    - Possíveis modelos para o processo de construção e montagem: *append*, exemplo do labirinto, árvores de *parsing* (nós retornados pelo *Builder* e utilizados em invocações futuras), etc
  - Classes abstratas para produtos ?
    - Geralmente os produtos criados por diferentes *Builders* são consideravelmente diferentes, tornando de pouca utilidade o uso de classes abstratas para produtos
    - O cliente sabe qual *Builder* usar e qual produto obter

Builder

- Implementação:
  - Métodos *default* vazios na interface do *Builder*:
    - Os métodos não são virtuais puros (abstratos) pois permite-se que os *Builders* concretos implementem somente aqueles métodos necessários

Builder

- Código exemplo:

```
class MazeBuilder {
public:
    virtual void BuildMaze() { }
    virtual void BuildRoom(int room) { }
    virtual void BuildDoor(int roomFrom, int roomTo) { }

    virtual Maze* GetMaze() { return 0; }
protected:
    MazeBuilder();
};
```

Builder

- Código exemplo:

```
Maze* MazeGame::CreateMaze (MazeBuilder& builder) {
    builder.BuildMaze();

    builder.BuildRoom(1);
    builder.BuildRoom(2);

    builder.BuildDoor(1, 2);

    return builder.GetMaze();
}
```

Builder

- Código exemplo:

```
Maze* MazeGame::CreateComplexMaze (MazeBuilder& builder) {
    builder.BuildRoom(1);
    // ...
    builder.BuildRoom(1001);

    return builder.GetMaze();
}
```

Builder

- Código exemplo:

```
class StandardMazeBuilder : public MazeBuilder {
public:
    StandardMazeBuilder();

    virtual void BuildMaze();
    virtual void BuildRoom(int);
    virtual void BuildDoor(int, int);

    virtual Maze* GetMaze();
private:
    Direction CommonWall(Room*, Room*);
    Maze* _currentMaze;
};
```

Builder

- Código exemplo:

```
void StandardMazeBuilder::BuildMaze () {
    _currentMaze = new Maze;
}
```

```
StandardMazeBuilder::StandardMazeBuilder () {
    _currentMaze = 0;
}
```

```
Maze* StandardMazeBuilder::GetMaze () {

    return _currentMaze;
}
```

Builder

- Código exemplo:

```
void StandardMazeBuilder::BuildRoom (int n) {
    if (!_currentMaze->RoomNo(n)) {
        Room* room = new Room(n);
        _currentMaze->AddRoom(room);

        room->SetSide(North, new Wall);
        room->SetSide(South, new Wall);
        room->SetSide(East, new Wall);
        room->SetSide(West, new Wall);
    }
}
```

Builder

- Código exemplo:

```
void StandardMazeBuilder::BuildDoor (int n1, int n2) {
    Room* r1 = _currentMaze->RoomNo(n1);
    Room* r2 = _currentMaze->RoomNo(n2);
    Door* d = new Door(r1, r2);

    r1->SetSide(CommonWall(r1,r2), d);
    r2->SetSide(CommonWall(r2,r1), d);
}
```

```
Maze* maze;
MazeGame game;
StandardMazeBuilder builder;

game.CreateMaze(builder);
maze = builder.GetMaze();
```

Builder

- Código exemplo:

```
class CountingMazeBuilder : public MazeBuilder {
public:
    CountingMazeBuilder();

    virtual void BuildMaze();
    virtual void BuildRoom(int);
    virtual void BuildDoor(int, int);
    virtual void AddWall(int, Direction);

    void GetCounts(int&, int&) const;
private:
    int _doors;
    int _rooms;
};
```

Builder

- Código exemplo:

```
CountingMazeBuilder::CountingMazeBuilder () {
    _rooms = _doors = 0;
}

void CountingMazeBuilder::BuildRoom (int) {
    _rooms++;
}

void CountingMazeBuilder::BuildDoor (int, int) {
    _doors++;
}

void CountingMazeBuilder::GetCounts (
    int& rooms, int& doors
) const {
    rooms = _rooms;
    doors = _doors;
}
```

Builder

- Código exemplo:

```
int rooms, doors;
MazeGame game;
CountingMazeBuilder builder;

game.CreateMaze(builder);
builder.GetCounts(rooms, doors);

cout << "The maze has "
     << rooms << " rooms and "
     << doors << " doors" << endl;
```

Builder

- Usos conhecidos:
  - Construção de árvores de análise sintática
  - Classes que funcionam ao mesmo tempo como *Builder* e *Director* para criar sub-classes delas mesmas

Builder

- Padrões relacionados:
  - *Abstract Factory*: também podem construir objetos complexos, porém com uma única operação
  - Geralmente o produto criado é um *Composite*

INF011 – Padrões de Projeto

04 – *Builder*

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