INF011 – Padrões de Projeto

21 – *Memento*

Sandro Santos Andrade
sandroandrade@ifba.edu.br

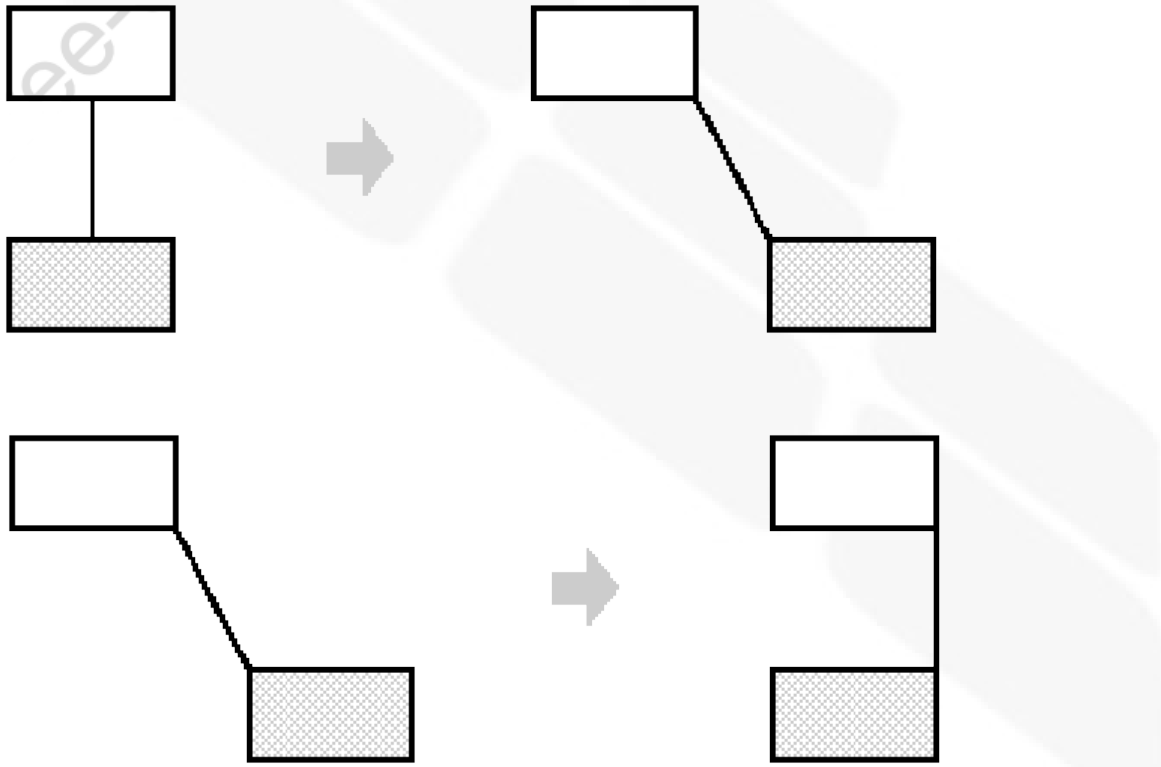
Instituto Federal de Educação, Ciência e Tecnologia da Bahia
Departamento de Tecnologia Eletro-Eletrônica
Graduação Tecnológica em Análise e Desenvolvimento de Sistemas

Memento

- Propósito:
  - Capturar e externalizar o estado interno de um objeto, sem violar o encapsulamento, com o objetivo de restaurar este estado no objeto em um momento futuro
- Também conhecido como: *Token*
- Motivação:
  - Pode-se precisar armazenar o estado interno de um objeto para implementar, por exemplo, *checkpoints* e mecanismos de *undo* que possibilitam o retrocesso da aplicação a um estado anterior
  - Expor o estado, entretanto, viola o encapsulamento e prejudica a confiabilidade e extensibilidade da aplicação

Memento

- Motivação:
  - Editor gráfico com conexões entre objetos (*ConstraintSolver*): processo falho de *undo*



Memento

- Motivação:
  - O *Memento* é um objeto que armazena um *snapshot* do estado interno de outro objeto – o **originador** do *Memento*
  - Processo correto de *undo* com uso do *Memento*:
    - O editor, como um efeito colateral da operação *move()*, requisita um *Memento* do *ConstraintSolver*
    - O *ConstraintSolver* cria e retorna o *Memento* (instância de *SolverState*), contendo estruturas de dados que descrevem o estado atual das equações e variáveis internas do *ConstraintSolver*
    - Mais tarde, quando o usuário desfizer a operação, o editor envia de volta o *SolverState* para o *ConstraintSolver*
    - Baseado na informação do *SolverState* o *ConstraintSolver* modifica suas estruturas internas para o estado anterior

Memento

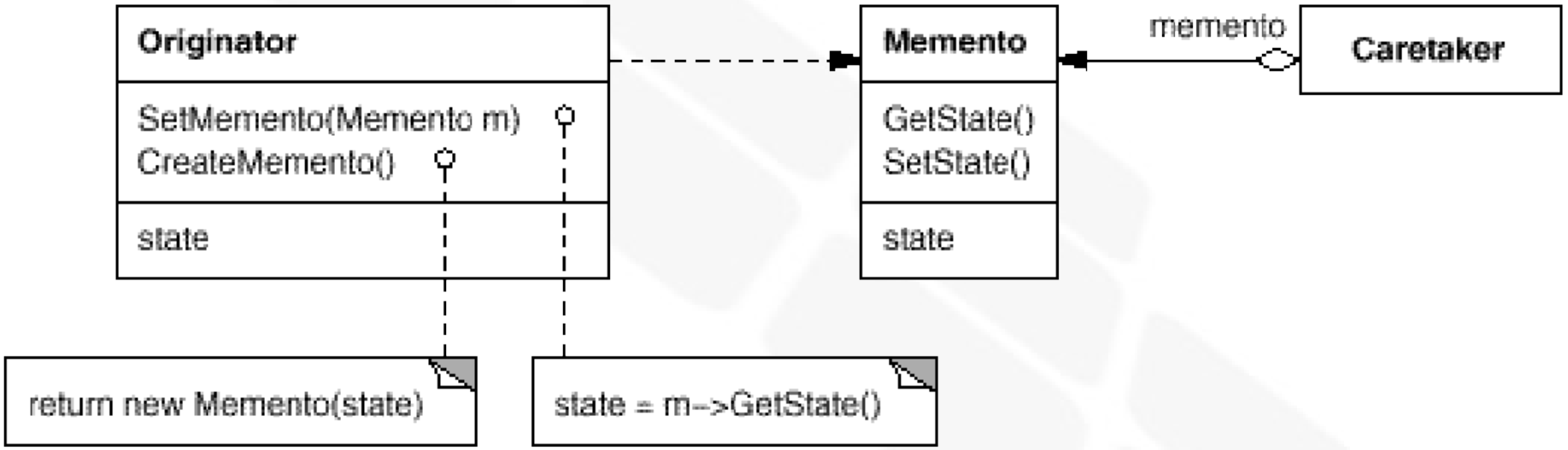
- Aplicabilidade:
  - Quando deseja-se salvar um *snapshot* do estado (ou parte do estado) de um objeto, de modo que ele possa ser restaurado para este estado, em um momento futuro

E

  - Utilizar uma interface direta para obter o estado iria expor detalhes de implementação e violar o encapsulamento do objeto

Memento

- Estrutura:



Memento

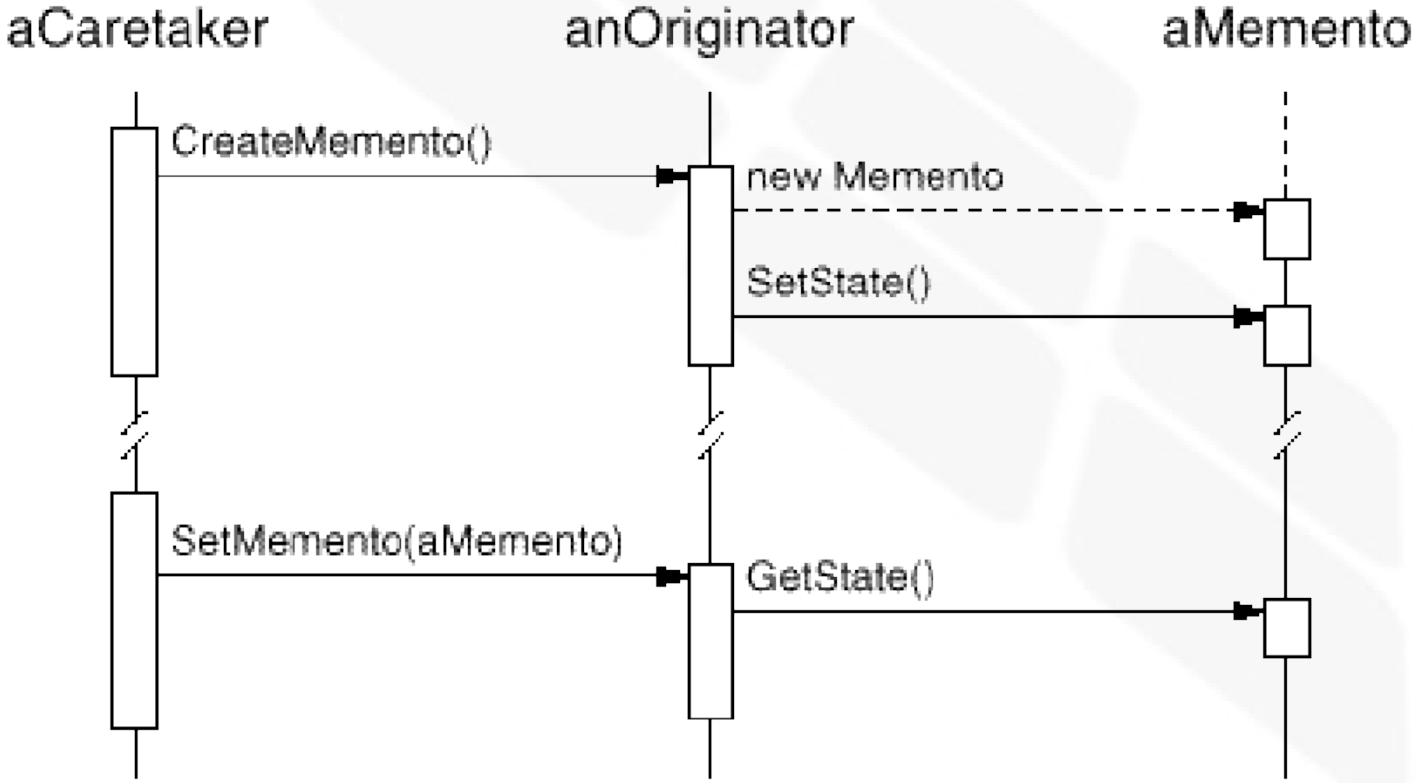
- Participantes:
  - *Memento* (SolverState):
    - Armazena o estado interno do objeto originador. Pode armazenar somente o estado mínimo necessário para a discretização do objeto
    - Impede o acesso de outro objetos que não sejam o originador
    - *Mementos* possuem duas interfaces:
      - Uma restrita (*narrow*), utilizada pelo *Caretaker*, somente para passar o *Memento* para outros objetos
      - Uma ampla (*wide*), utilizada pelo originador, para acessar todos os dados necessários à restauração do estado prévio

Memento

- Participantes:
  - *Originator* (ConstraintSolver):
    - Cria o *Memento* contendo o *snapshot* do estado interno atual
    - Usa o *Memento* para restaurar seu estado interno
  - *Caretaker* (mecanismo de *undo*):
    - É responsável pela custódia do *Memento*
    - Nunca opera o *Memento* ou examina o seu conteúdo

Memento

- Colaborações:
  - O *Caretaker* solicita o *Memento* ao originador, o mantém por um tempo e então o envia de volta ao originador



Memento

- Consequências:
  - Preserva os limites de encapsulamento:
    - O *Memento* evita expor informações que somente o originador deveria manipular mas precisam ser armazenadas fora do originador
    - Isola outros objetos de detalhes internos potencialmente complexos, preservando os limites do encapsulamento
  - Simplifica o originador:
    - Em outras soluções o originador manteria versões de estados internos requisitados pelos clientes
    - Isto traria toda a responsabilidade de gerenciamento deste armazenamento para o originador
    - Ao decidir que os clientes gerenciarão o estado mantém-se o originador simples

Memento

- Consequências:
  - O uso do *Memento* pode gerar altos custos:
    - Alto *overhead* pode ser gerado se o originador tiver de copiar grandes quantidades de informação para o *Memento* ou se os clientes criam e retornam *Mementos* para o originador muito frequentemente
    - O padrão só é apropriado quando o encapsulamento e restauração do estado do originador têm custo baixo
  - Definindo as interfaces *narrow* e *wide*:
    - Pode ser difícil em algumas linguagens garantir que somente o originador irá acessar o estado do *Memento*
  - Custos adicionais de custódia de *Mementos*:
    - O *Caretaker* é responsável pela desalocação de *Mementos* porém ele não sabe o tamanho do *Memento*

Memento

- Implementação:
  - Suporte da linguagem:
    - Para implementar as interfaces *narrow* e *wide* a linguagem de programação deveria, idealmente, suportar dois níveis de proteção estática:

```
class State;

class Originator {
public:
    Memento* CreateMemento();
    void SetMemento(const Memento*);
    // ...
private:
    State* _state;      // internal data structures
    // ...
};
```

Memento

- Implementação:
  - Suporte da linguagem:
    - Para implementar as interfaces *narrow* e *wide* a linguagem de programação deveria, idealmente, suportar dois níveis de proteção estática:

```
class Memento {
public:
    // narrow public interface
    virtual ~Memento();
private:
    // private members accessible only to Originator
    friend class Originator;
    Memento();

    void SetState(State*);
    State* GetState();
    // ...
private:
    State* _state;
    // ...
};
```

Memento

- Implementação:
  - Armazenando mudanças incrementais:
    - Quando os *Mementos* são criados e devolvidos para o originador, em uma sequência prevista, o *Memento* pode armazenar somente a mudança incremental do estado do originador
    - Ex: comandos *undoable* podem utilizar *Mementos* para garantir que comandos sejam restaurados para o seu estado anterior quando forem desfeitos

Memento

- Código exemplo:

```
class Graphic;
    // base class for graphical objects in the graphical editor

class MoveCommand {
public:
    MoveCommand(Graphic* target, const Point& delta);
    void Execute();
    void Unexecute();
private:
    ConstraintSolverMemento* _state;
    Point _delta;
    Graphic* _target;
};
```

Memento

- Código exemplo:

```
class ConstraintSolver {
public:
    static ConstraintSolver* Instance();

    void Solve();
    void AddConstraint(
        Graphic* startConnection, Graphic* endConnection
    );
    void RemoveConstraint(
        Graphic* startConnection, Graphic* endConnection
    );

    ConstraintSolverMemento* CreateMemento();
    void SetMemento(ConstraintSolverMemento*);
private:
    // nontrivial state and operations for enforcing
    // connectivity semantics
};
```

Memento

- Código exemplo:

```
class ConstraintSolverMemento {
public:
    virtual ~ConstraintSolverMemento();
private:
    friend class ConstraintSolver;
    ConstraintSolverMemento();

    // private constraint solver state
};
```

```
void MoveCommand::Execute () {
    ConstraintSolver* solver = ConstraintSolver::Instance();
    _state = solver->CreateMemento(); // create a memento
    _target->Move(_delta);
    solver->Solve();
}
```

```
void MoveCommand::Unexecute () {
    ConstraintSolver* solver = ConstraintSolver::Instance();
    _target->Move(-_delta);
    solver->SetMemento(_state); // restore solver state
    solver->Solve();
}
```

Memento

- Usos conhecidos:
  - *UniDraw*
  - Dylan
  - QOCA

Memento

- Padrões relacionados:
  - *Commands* podem utilizar *Mementos* para manter o estado de operações reversíveis
  - *Mementos* podem ser utilizados para iterações

INF011 – Padrões de Projeto

21 – *Memento*

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