INF011 – Padrões de Projeto

06 – *Prototype*

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

Prototype

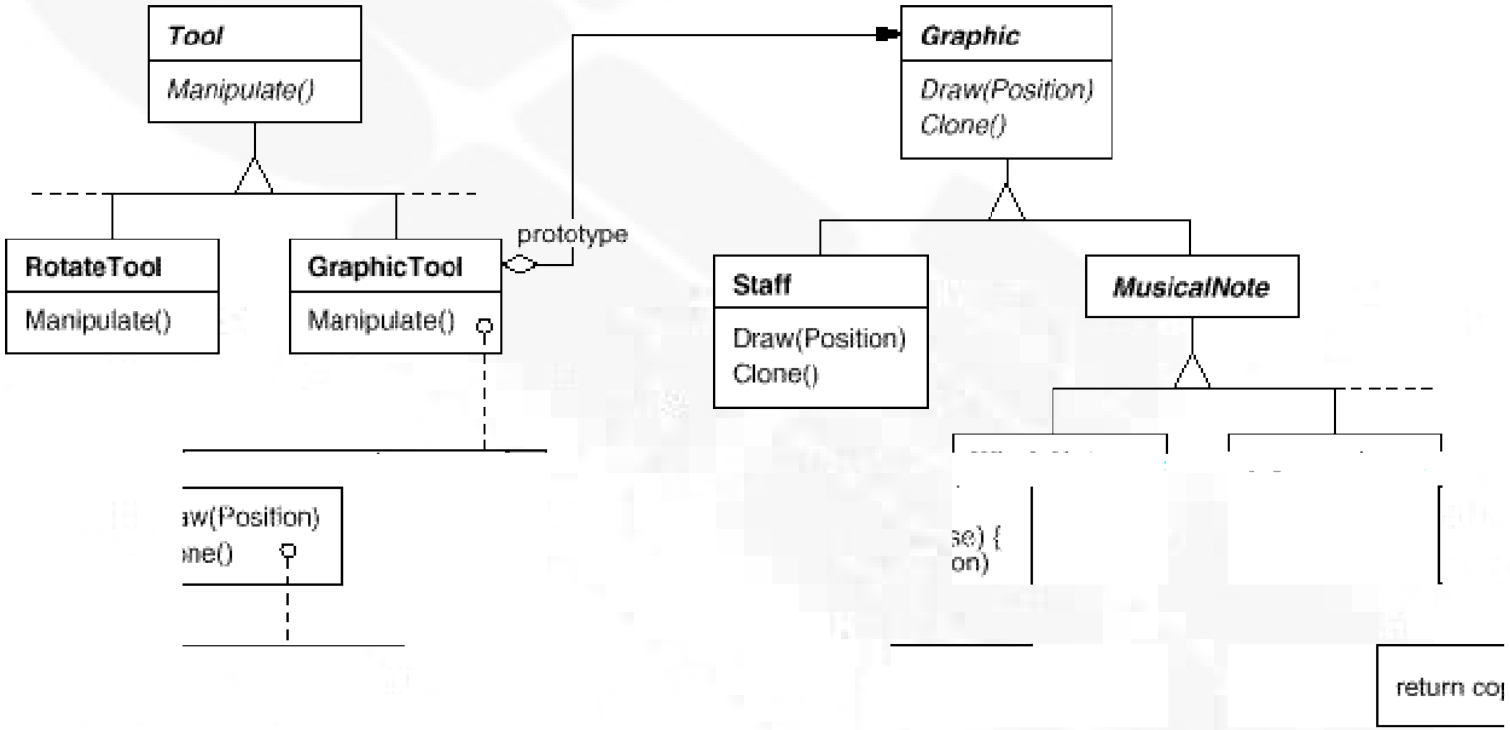
- Propósito:
  - Especificar, através de uma instância-protótipo, os tipos de objetos a serem criados. Novos objetos são criados copiando os protótipos
- Motivação:
    - Criação de um editor de partituras musicais a partir de um *framework* genérico para editores gráficos
    - Editor = *toolbar* de ferramentas para inserção de elementos musicais + *toolbar* de ferramentas para manipulação (mover, remover, etc) desses elementos
    - Classes abstratas do *framework*: *Graphic* e *Tool*
    - A sub-classe *GraphicTool* do *framework* cria instâncias de elementos musicais e os adiciona à partitura

Prototype

- Motivação:
  - As classes que representam notas e claves são específicas da aplicação e não devem estar presentes em *GraphicTool*
  - Pode-se derivar *GraphicTool* para cada tipo de elemento musical mas muitas sub-classes seriam criadas (composição é melhor que herança)

Prototype

- Motivação:



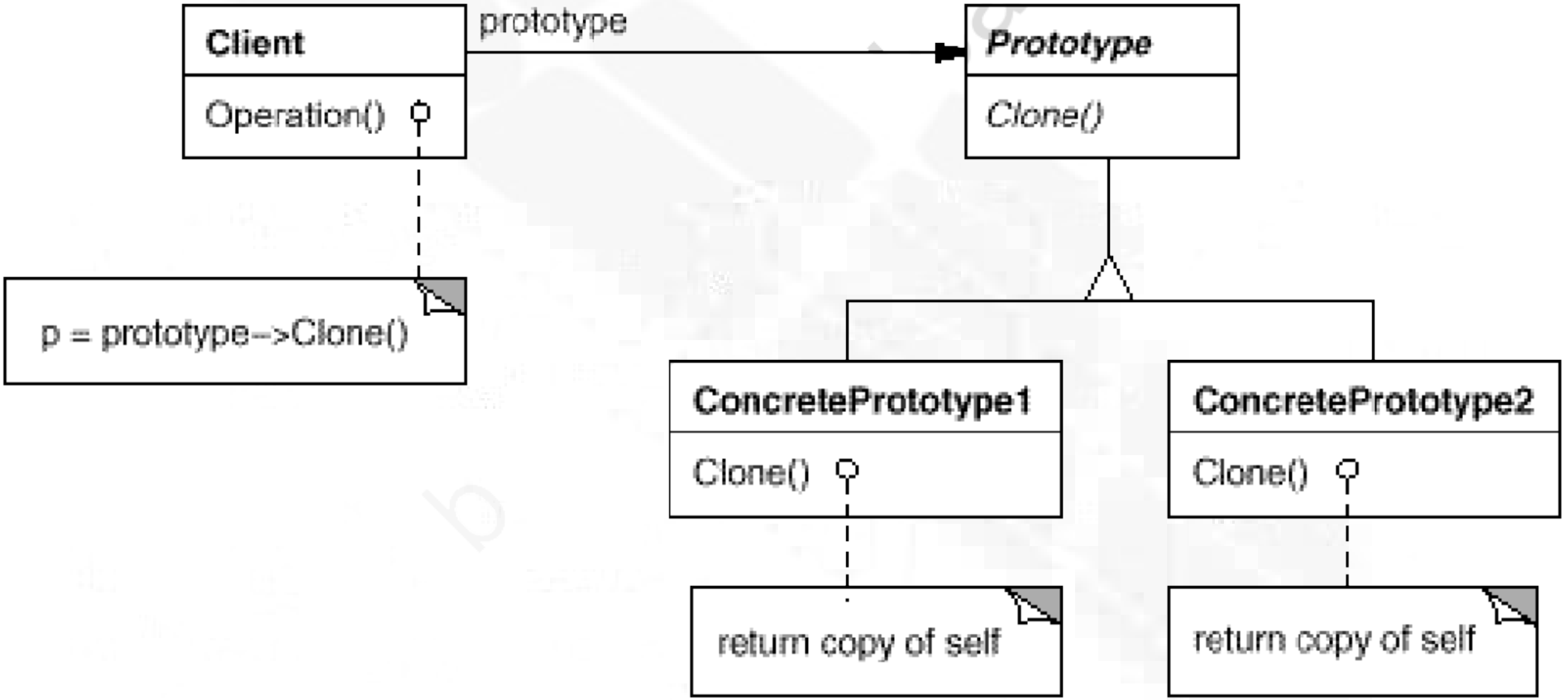
  - Pode-se reduzir ainda mais o número de classes: *WholeNote* e *HalfNote* podem ser uma só

Prototype

- Aplicabilidade:
  - O sistema deve ser independente de coma OA6m Au de j cor t

Prototype

- Estrutura:



Prototype

- Participantes:
  - *Prototype* (Graphics): declara uma interface para se auto-duplicar
  - *ConcretePrototype* (Staff, WholeNote e HalfNote): implementa a operação de auto-duplicação
  - *Client* (GraphicTool): cria um novo objeto solicitando que o *Prototype* se duplique

Prototype

- Colaborações:
  - O cliente solicita ao *Prototype* que se auto-duplique

Prototype

- Consequências:
  - Compartilha muitas das consequências do *Abstract Factory* e *Builder*:
    - Isola as classes concretas do cliente
    - Permite que clientes trabalhem com classes específicas da aplicação sem requerer modificações
  - Adição e remoção de produtos em *run-time*
  - Especificação de novos objetos através da variação de valores:
    - Sistemas altamente dinâmicos permitem a definição de novos comportamentos através de composição
    - O objeto agregado pode ser criado via *Prototype*
    - Define-se “novas classes” em *run*-time, sem programação

Prototype

- Consequências:
  - Especificação de novos objetos através da variação de estrutura:
    - Muitas aplicações constroem objetos a partir de partes e sub-partes (ex: circuitos e sub-circuitos)
    - O sub-circuito pode ser um *Prototype* para o *toolbar* de elementos de circuito
    - Se o circuito (*composite*) também implementar o método de duplicação (com *deep copy*) então circuitos com diferentes estruturas podem ser tornar *Prototypes*
  - Número reduzido de sub-classes
  - Permite configuração dinâmica das classes utilizadas na aplicação (meta-classes em Java)
  - A implementação do método de duplicação pode ser complicada (*deep copy*, referências cruzadas, etc)

Prototype

- Implementação:
  - O *Prototype* é mais útil em linguagens “estáticas”, onde classes não são objetos (C++) e quando existe pouca ou nenhuma informação de tipo em *run-time*
  - Uso de um *Prototype Manager*:
    - Quando os protótipos de um sistema podem ser criados e destruídos dinamicamente
    - Os clientes não gerenciam os protótipos diretamente, ao invés solicitam seu armazenamento e recuperação ao *Prototype Manager* (registro)
    - Os protótipos são identificados, no registro, por uma chave

Prototype

- Implementação:
  - Implementação do método de duplicação:
    - Parte mais difícil
    - *Shallow Copy* x *Deep Copy* (exs: Smalltalk e C++)
    - Geralmente requer *deep copy*
    - Se os objetos possuem operação de *save* e *load*, pode-se salvá-lo em um *buffer* de memória e então realizar uma cópia desta região
  - Inicializando clones:
    - Alguns clones precisam ser inicializados com valores específicos, que não podem ser fornecidos na operação de duplicação
    - Utiliza-se um método *initialize*, com parâmetros específicos

Prototype

- Código exemplo:

```
class MazePrototypeFactory : public MazeFactory {
public:
    MazePrototypeFactory(Maze*, Wall*, Room*, Door*);

    virtual Maze* MakeMaze() const;
    virtual Room* MakeRoom(int) const;
    virtual Wall* MakeWall() const;
    virtual Door* MakeDoor(Room*, Room*) const;

private:
    Maze* _prototypeMaze;
    Room* _prototypeRoom;
    Wall* _prototypeWall;
    Door* _prototypeDoor;
};
```

Prototype

- Código exemplo:

```
MazePrototypeFactory::MazePrototypeFactory (
    Maze* m, Wall* w, Room* r, Door* d
) {
    _prototypeMaze = m;
    _prototypeWall = w;
    _prototypeRoom = r;
    _prototypeDoor = d;
}
```

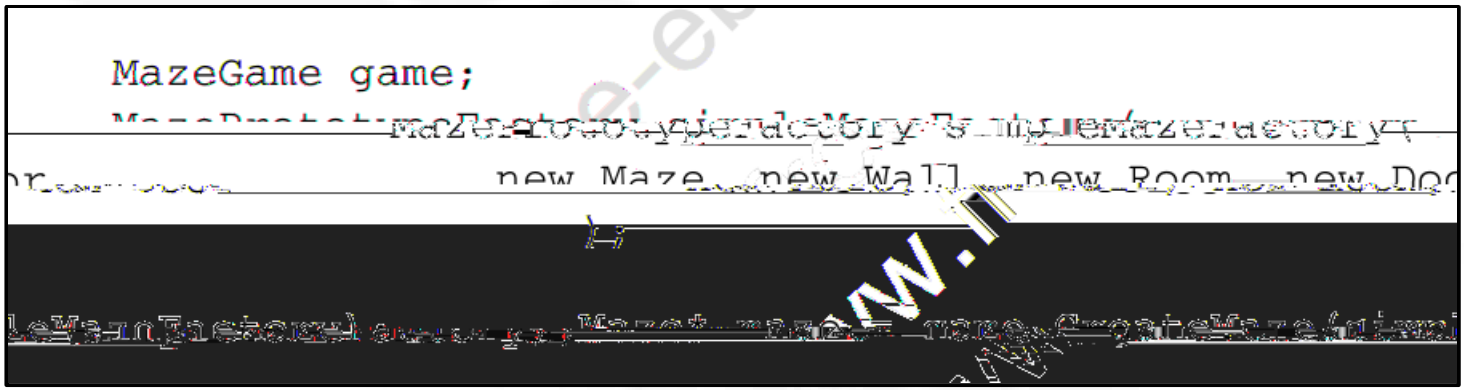
```
Wall* MazePrototypeFactory::MakeWall () const {
    return _prototypeWall->Clone();
}

Door* MazePrototypeFactory::MakeDoor (Room* r1, Room *r2) const {
    Door* door = _prototypeDoor->Clone();
    door->Initialize(r1, r2);
    return door;
}
```

ks-download.org

Prototype

- Código exemplo:




```
MazePrototypeFactory bombedMazeFactory(
    new Maze, new BombedWall,
    new RoomWithABomb, new Door
);
```

Prototype

- Código exemplo:

```
class Door : public MapSite {
public:
    Door();
    Door(const Door&);

    virtual void Initialize(Room*, Room*);
    virtual Door* Clone() const;
    virtual void Enter();
    Room* OtherSideFrom(Room*);
private:
    Room* _room1;
    Room* _room2;
};
```

Prototype

- Código exemplo:

```
Door::Door (const Door& other) {
    _room1 = other._room1;
    _room2 = other._room2;
}

void Door::Initialize (Room* r1, Room* r2) {
    _room1 = r1;
    _room2 = r2;
}

Door* Door::Clone () const {
    return new Door(*this);
}
```

Prototype

- Código exemplo:

```
class BombedWall : public Wall {
public:
    BombedWall();
    BombedWall(const BombedWall&);

    virtual Wall* Clone() const;
    bool HasBomb();
private:
    bool _bomb;
};

BombedWall::BombedWall (const BombedWall& other) : Wall(other) {
    _bomb = other._bomb;
}

Wall* BombedWall::Clone () const {
    return new BombedWall(*this);
}
```

www.free-ebooks-downloa

Prototype

- Usos conhecidos:
  - *ThingLab*:
    - Usuários podem criar um objeto *composite* e então promovê-lo a protótipo, realizando sua inclusão em uma biblioteca de objetos reutilizáveis
  - Dentre outros

Prototype

- Padrões relacionados:
  - *Prototype* e *Abstract Factory* são concorrentes
  - Entretanto podem ser utilizados em conjunto
  - Projetos com uso intensivo dos padrões *Composite* e *Decorator* geralmente se beneficiam do uso do *Prototype*

INF011 – Padrões de Projeto

06 – *Prototype*

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