INF011 – Padrões de Projeto

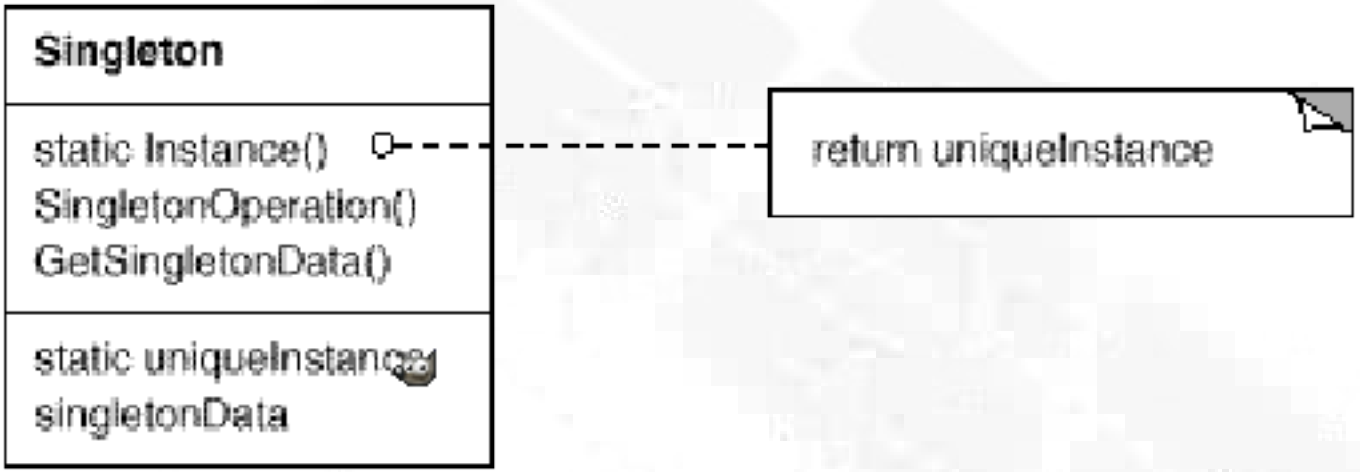
07 – *Singleton*

sandroandrade@ifba.edu.br

- Propósito:
  - Garantir que uma classe tenha somente uma única instância e disponibilizar um ponto global de acesso a ela
- Motivação:
    - Um sistema pode utilizar muitas impressoras mas terá somente um *spooler* de impressão
    - O sistema deve utilizar somente um sistema de arquivos e um *window manager*
    - Uma variável global torna o objeto acessível mas não previne a criação de múltiplas instâncias
    - A solução é fazer com que a própria classe cuide desta instância única e garanta que uma segunda não será criada. Provê também uma forma para acessar a instância

- Aplicabilidade:
  - Deve existir somente uma instância da classe e os clientes devem utilizá-la através de um ponto de acesso bem definido
  - Esta instância única deve poder ser extendida e os clientes devem poder usar a versão extendida sem modificar os seus códigos-fonte

- Estrutura:


Singleton
static Instance()
SingletonOperation()
GetSingletonData()
static uniqueInstance
singletonData
return uniqueInstance

- Participantes:
  - *Singleton*
    - Define a operação *instance()* que permite a clientes o acesso à instância única. *Instance()* é um método estático
    - Pode ser responsável por criar a sua instância única

- Colaborações:
  - Clientes acessam a instância única do *Singleton* somente através do método *instance()* do *Singleton*

- Consequências:
  - Acesso controlado à instância única:
    - A própria classe encapsula sua instância única e, portanto, tem estrito controle sobre como e quando os clientes a acessam
  - Redução no *namespace*:
    - O *Singleton* é uma alternativa eficiente às variáveis globais. Evita poluir o s

- Consequências:
  - Permite um número variado de instâncias:
    - Pode permitir a criação de mais de uma instância do *Singleton* ou até controlar o número de instâncias existentes. Somente a operação *instance()* precisa ser modificada
  - É mais flexível que métodos e atributos estáticos:
    - Com métodos e atributos estáticos é difícil permitir que mais de uma instância exista
    - Métodos estáticos em C++ não podem ser virtuais e portanto sub-classes não podem sobrescrevê-los

- Implementação:
  - Garantindo uma única instância:
    - A classe é criada de modo que somente uma instância possa ser criada
    - Geralmente oculta-se a operação que cria a instância dentro de um método estático – *instance()* – que garante que somente uma instância será criada

- Implementação:

```
class Singleton {
public:
    static Singleton* Instance();
protected:
    Singleton();
private:
    static Singleton* _instance;
};
```

```
Singleton* Singleton::_instance = 0;

Singleton* Singleton::Instance () {
    if (_instance == 0) {
        _instance = new Singleton;
    }
    return _instance;
}
```

- Implementação:
  - Note que o construtor é *protected*. Se chamado diretamente irá gerar um erro de compilação. Isto garante que realmente só uma instância será criada
  - O método *instance()* pode atribuir ao ponteiro *_instance* o endereço de uma instância de uma de *Singleton*
  - Outras desvantagens de usar atributos ou métodos estáticos:
    - Não impede que o programador crie mais de uma instância
    - Pode-se não ter todas as informações necessárias para inicializar o *Singleton* no momento da inicialização estática
    - O C++ não define a ordem de construção de objetos globais localizados em unidades de tradução diferentes
    - Todos os *Singletons* são criados, sejam eles usados no programa ou não

- Implementação:
  - Derivando a classe *Singleton*:
    - O problema não é implementar a sub-classe mas configurar a aplicação para utilizá-la
    - A forma mais simples é modificar o método *instance()*
    - Outra solução é mover a implementação do método *instance()* da classe pai para as classes filhas e decidir qual *Singleton* utilizar em tempo de *link*. Não é possível escolher o *Singleton* em *run-time*, entretanto
    - Pode-se utilizar condicionais para determinar a sub-classe a ser instanciada porém o conjunto de alternativas está *hard-coded*
    - Uma abordagem mais flexível é utilizar um ***singletons***

- Implementação:
  - Derivando a classe *Singleton* (registro de *singletons*):
    - Ao invés do método *instance()* decidir qual *singleton* utilizar as diferentes classes *Singleton* registram sua instância única, identificadas por um nome, em um registro bem conhecido
    - Quando o método *instance()* precisa de um *singleton* ele consulta o registro pelo nome desejado
    - Tudo o que é necessário é uma interface, comum a todos os *singletons*, contendo as operações do registro

- Implementação:
  - Derivando a classe *Singleton* (registro de *singletons*):

```
class Singleton {
public:
    static void Register(const char* name, Singleton*);
    static Singleton* Instance();
protected:
    static Singleton* Lookup(const char* name);
private:
    static Singleton* _instance;
    static List<NameSingletonPair>* _registry;
};
```

- Implementação:
  - Derivando a classe *Singleton* (registro de *singletons*):

```
Singleton* Singleton::Instance () {
    if (_instance == 0) {
        const char* singletonName = getenv("SINGLETON");
        // user or environment supplies this at startup

        _instance = Lookup(singletonName);
        // Lookup returns 0 if there's no such singleton
    }
    return _instance;
}
```

- Implementação:
  - Como os diversos *singleton* se registram ?

```
MySingleton::MySingleton() {
    // ...
    Singleton::Register("MySingleton", this);
}
```

```
static MySingleton theSingleton;
```

- Código exemplo:

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

    // existing interface goes here
protected:
    MazeFactory();
private:
    static MazeFactory* _instance;
};
```

- Código exemplo:

```
MazeFactory* MazeFactory::_instance = 0;

MazeFactory* MazeFactory::Instance () {
    if (_instance == 0) {
        _instance = new MazeFactory;
    }
    return _instance;
}
```

- Código exemplo:

```
MazeFactory* MazeFactory::Instance () {
    if (_instance == 0) {
        const char* mazeStyle = getenv("MAZESTYLE");

        if (strcmp(mazeStyle, "bombed") == 0) {
            _instance = new BombedMazeFactory;

        } else if (strcmp(mazeStyle, "enchanted") == 0) {
            _instance = new EnchantedMazeFactory;

        // ... other possible subclasses

        } else {        // default
            _instance = new MazeFactory;
        }
    }
    return _instance;
}
```

- Usos conhecidos:
  - Relacionamento classes – metaclasses em Java
  - O *InterView* usa o *singleton* para a sessão e a fábrica de *widgets* sendo utilizada

- Padrões relacionados:
  - Muitos padrões podem ser implementados usando o *Singleton*: *Abstract Factory, Builder, Prototype*

INF011 – Padrões de Projeto

07 – *Singleton*

sandroandrade@ifba.edu.br