INF011 – Padrões de Projeto

10 – *Bridge*

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

Bridge

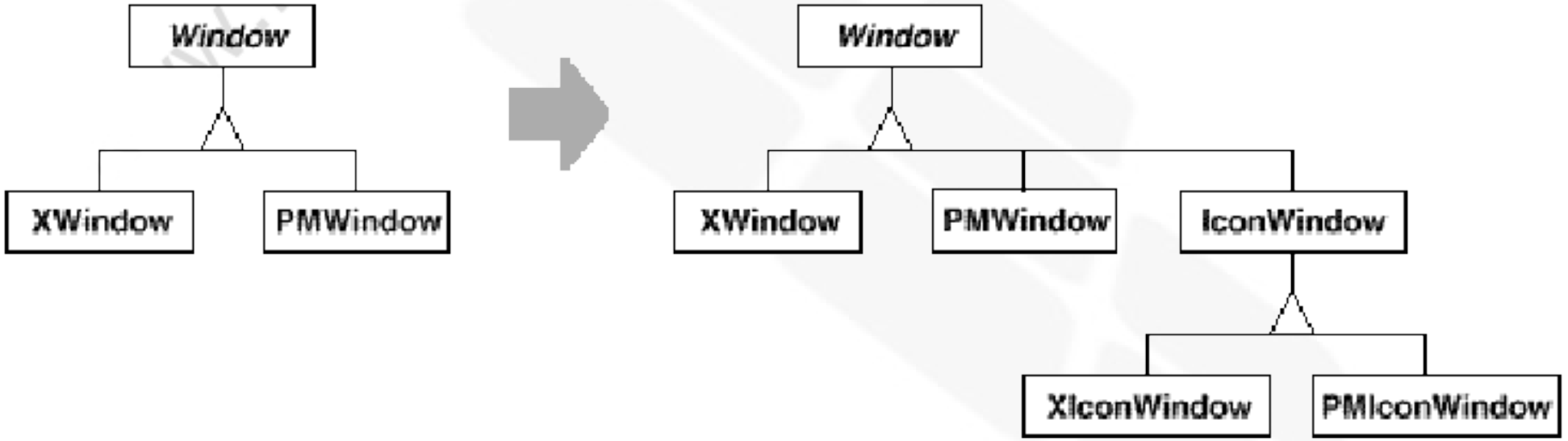
- Propósito:
  - Desacoplar uma abstração de sua implementação de modo que os dois possam variar de forma independente
- Também conhecido como: *Handle/Body*
- Motivação:
  - Quando uma abstração pode ter várias implementações geralmente define-se uma interface e deriva-se classes concretas com diferentes implementações
  - Entretanto, a herança define uma dependência permanente, tornando difícil modificar, estender e reutilizar abstrações e implementações de forma independente

Bridge

- Motivação:
  - Exemplo: implementação da classe *Window* de um *toolkit* portável para interfaces gráficas de usuário
  - As aplicações devem funcionar tanto no *X Window* quanto no *Presentation Manager*
  - 1ª solução: classe abstrata *Window* e sub-classes *XWindow* e *PMWindow*

Bridge

- Motivação:
  - 1ª solução: problemas



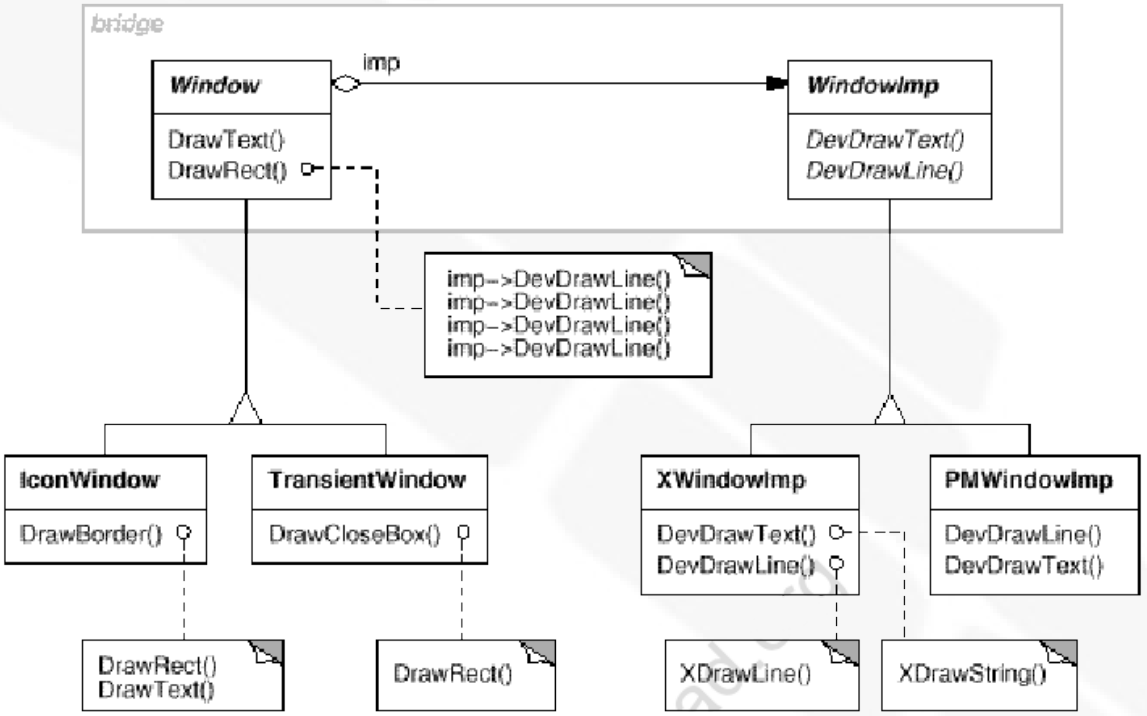
  - Grande número de classes

Bridge

- Motivação:
  - 1ª solução: problemas
    - O código do cliente é dependente de plataforma. Sempre que for necessário criar uma janela uma classe concreta com implementação específica será instanciada:
    - Window *w = new XWindow;
    - Clientes devem poder criar uma janela sem nenhum comprometimento com implementações concretas

Bridge

- Motivação:



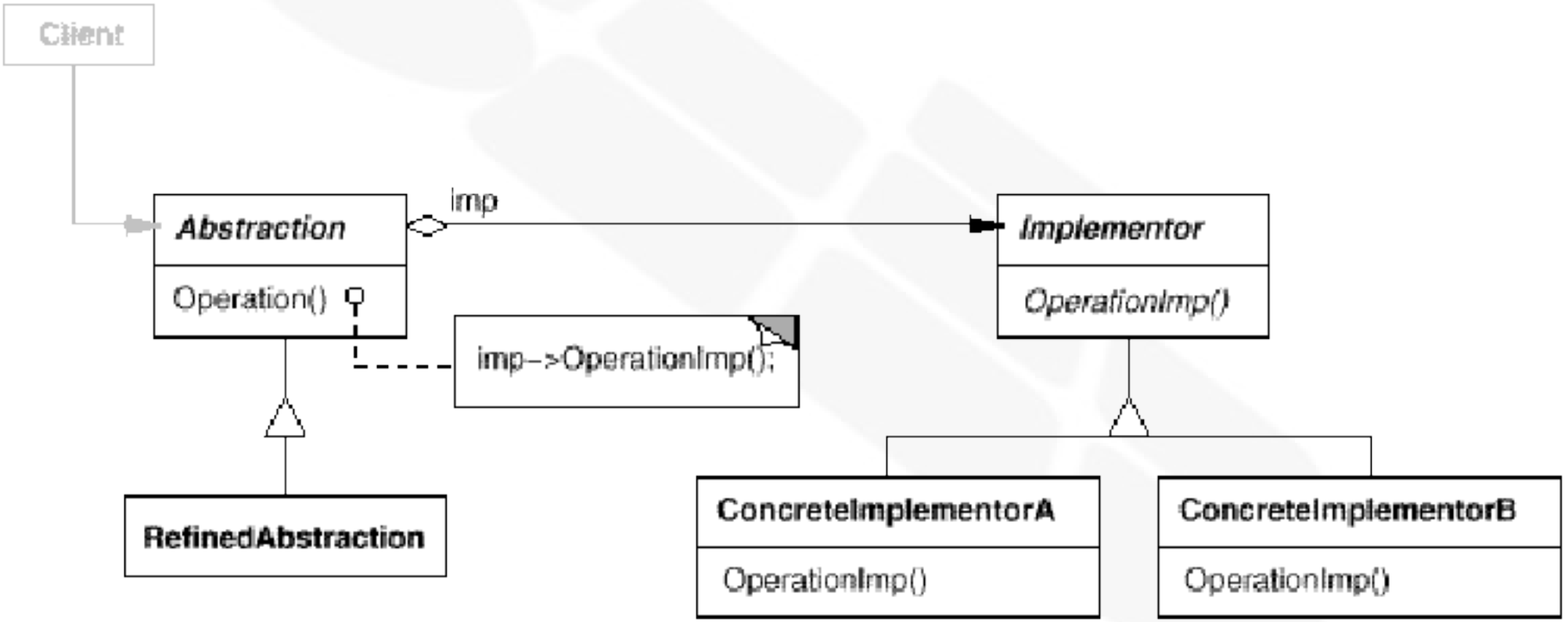
  - Todas as operações das sub-classes de *Window* são implementadas em termos de operações abstratas da interface *WindowsImpl*

Bridge

- Aplicabilidade:
  - Deseja-se evitar uma ligação permanente entre uma abstração e sua implementação
  - Deseja-se estender tanto a abstração quanto a implementação através de sub-classes
  - Deseja-se que mudanças na implementação da abstração não gerem impactos nos clientes
  - Quando existe uma proliferação de classes, geralmente indicando a necessidade de divisão de um objeto em dois
  - Deseja-se compartilhar uma implementação entre múltiplos objetos e este fato deve estar escondido dos clientes

Bridge

- Estrutura:



Bridge

- Participantes:
  - *Abstraction* (Window): define a interface da abstração e mantém uma referência para o objeto da implementação
  - *RefinedAbstraction* (IconWindow): estende a interface definida por *Abstraction*
  - *Implementor* (WindowImpl): define a interface das classes de implementação. Geralmente esta interface contém somente operações primitivas, enquanto a interface da abstração define operações de nível mais alto, baseadas nas primitivas da interface de implementação
  - *ConcreteImplementor* (XWindowImpl, PMWindowImpl): implementa a interface do *Implementor* e define sua implementação concreta

Bridge

- Colaborações:
  - O *Abstraction* repassa as requisições dos clientes para o objeto *Implementor*

Bridge

- Consequências:
  - Desacopla interface da implementação:
    - A implementação não fica mais permanentemente amarrada à interface. A implementação da abstração pode ser informada em *run-time*
    - Também elimina dependências em tempo de compilação: pode-se mudar a classe de implementação sem requerer a recompilação da classe de abstração e seus clientes
  - Melhora a extensibilidade:
    - Pode-se estender as hierarquias *Abstraction* e *Implementor* de forma independente
  - Esconde detalhes de implementação dos clientes

Bridge

- Implementação:
  - Somente um *Implementor*: neste caso não é necessário o *Implementor* abstrato. Caso degenerado do *Bridge*: um-para-um. Ainda é útil pois mudanças na implementação exigem apenas a operação de *link* sem recompilação
  - Onde criar o objeto *Implementor* ?
    - Se *Abstraction* conhece todas as implementações concretas ele pode decidir qual implementação utilizar baseado, por exemplo, em um parâmetro do construtor
    - Pode-se utilizar uma implementação *default* e trocá-la em *run-time* quando necessário
    - Pode-se também utilizar uma fábrica

Bridge

- Implementação:
  - Compartilhando implementações:

```
Handle& Handle::operator= (const Handle& other)  {
    other._body->Ref();
    _body->Unref();

    if (_body->RefCount() == 0) {
        delete _body;
    }
    _body = other._body;

    return *this;
}
```

Bridge

- Código exemplo:

```
class Window {
public:
    Window(View* contents);

    // requests handled by window
    virtual void DrawContents();

    virtual void Open();
    virtual void Close();
    virtual void Iconify();
    virtual void Deiconify();

    // requests forwarded to implementation
    virtual void SetOrigin(const Point& at);
    virtual void SetExtent(const Point& extent);
    virtual void Raise();
    virtual void Lower();

    virtual void DrawLine(const Point&, const Point&);
    virtual void DrawRect(const Point&, const Point&);
    virtual void DrawPolygon(const Point[], int n);
    virtual void DrawText(const char*, const Point&);

protected:
    WindowImp* GetWindowImp();
    View* GetView();

private:
    WindowImp* _imp;
    View* _contents; // the window's contents
};
```

Bridge

- Código exemplo:

```
class WindowImp {
public:
    virtual void ImpTop() = 0;
    virtual void ImpBottom() = 0;
    virtual void ImpSetExtent(const Point&) = 0;
    virtual void ImpSetOrigin(const Point&) = 0;

    virtual void DeviceRect(Coord, Coord, Coord, Coord) = 0;
    virtual void DeviceText(const char*, Coord, Coord) = 0;
    virtual void DeviceBitmap(const char*, Coord, Coord) = 0;
    // lots more functions for drawing on windows...
protected:
    WindowImp();
};
```

Bridge

- Código exemplo:

```
class ApplicationWindow : public Window {
public:
    // ...
    virtual void DrawContents();
};

void ApplicationWindow::DrawContents () {
    GetView()->DrawOn(this);
}
```

```
class IconWindow : public Window {
public:
    // ...
    virtual void DrawContents();
private:
    const char* _bitmapName;
};
```

Bridge

- Código exemplo:

```
void IconWindow::DrawContents() {
    WindowImp* imp = GetWindowImp();
    if (imp != 0) {
        imp->DeviceBitmap(_bitmapName, 0.0, 0.0);
    }
}
```

```
void Window::DrawRect (const Point& p1, const Point& p2) {
    WindowImp* imp = GetWindowImp();
    imp->DeviceRect(p1.X(), p1.Y(), p2.X(), p2.Y());
}
```

Bridge

- Código exemplo:

```
class XWindowImp : public WindowImp {
public:
    XWindowImp();

    virtual void DeviceRect(Coord, Coord, Coord, Coord);
    // remainder of public interface...
private:
    // lots of X window system-specific state, including:
    Display* _dpy;
    Drawable _winid;  // window id
    GC _gc;           // window graphic context
};
```

Bridge

- Código exemplo:

```
class PMWindowImp : public WindowImp {
public:
    PMWindowImp();
    virtual void DeviceRect(Coord, Coord, Coord, Coord);

    // remainder of public interface...
private:
    // lots of PM window system-specific state, including:
    HPS _hps;
};
```

Bridge

- Usos conhecidos:
  - ET++
  - libg++
  - NeXT's AppKit

Bridge

- Padrões relacionados:
  - Um *Abstract Factory* pode criar e configurar um *Bridge* particular
  - O *Adapter* tem como objetivo fazer com que classes não-relacionadas trabalhem em conjunto. Geralmente são utilizados depois que o sistema foi projetado
  - O *Bridge*, por outro lado, é já aplicado durante o projeto do sistema, com o objetivo de permitir que abstrações e implementações variem independentemente

INF011 – Padrões de Projeto

10 – *Bridge*

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