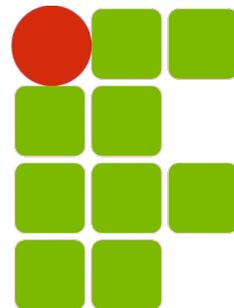


INF011 – Padrões de Projeto

22 – *Observer*

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



**INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA**

Observer

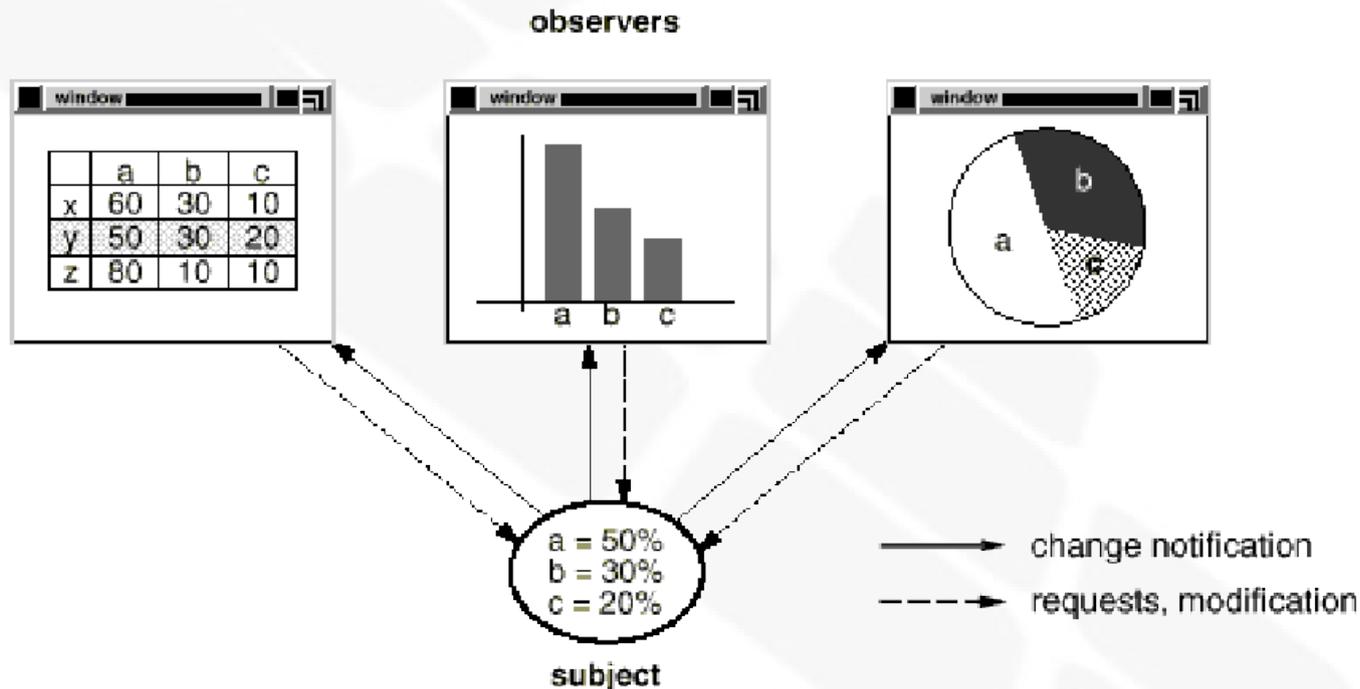
- Propósito:
 - Definir uma dependência do tipo um-para-muitos entre objetos, de modo que quando um objeto mudar seu estado todos os seus dependentes serão notificados e atualizados automaticamente
- Também conhecido como: *Dependents*, *Publish-Subscribe*
- Motivação:
 - Um efeito colateral comum de particionar o sistema em uma coleção de classes interoperantes é a manutenção da consistência entre objetos relacionados

Observer

- Motivação:
 - A consistência não deve ser mantida através do alto acoplamento entre classes pois isso reduz seu potencial de reuso
 - Ex: classes que definem dados da aplicação e suas apresentações podem ser reutilizadas de forma independente
 - Ex: um objeto de planilha de cálculo e outro para gráficos de barra exibem a mesma informação em apresentações diferentes
 - Embora a planilha e o gráfico não se conheçam diretamente eles se comportam como tal

Observer

- Motivação:



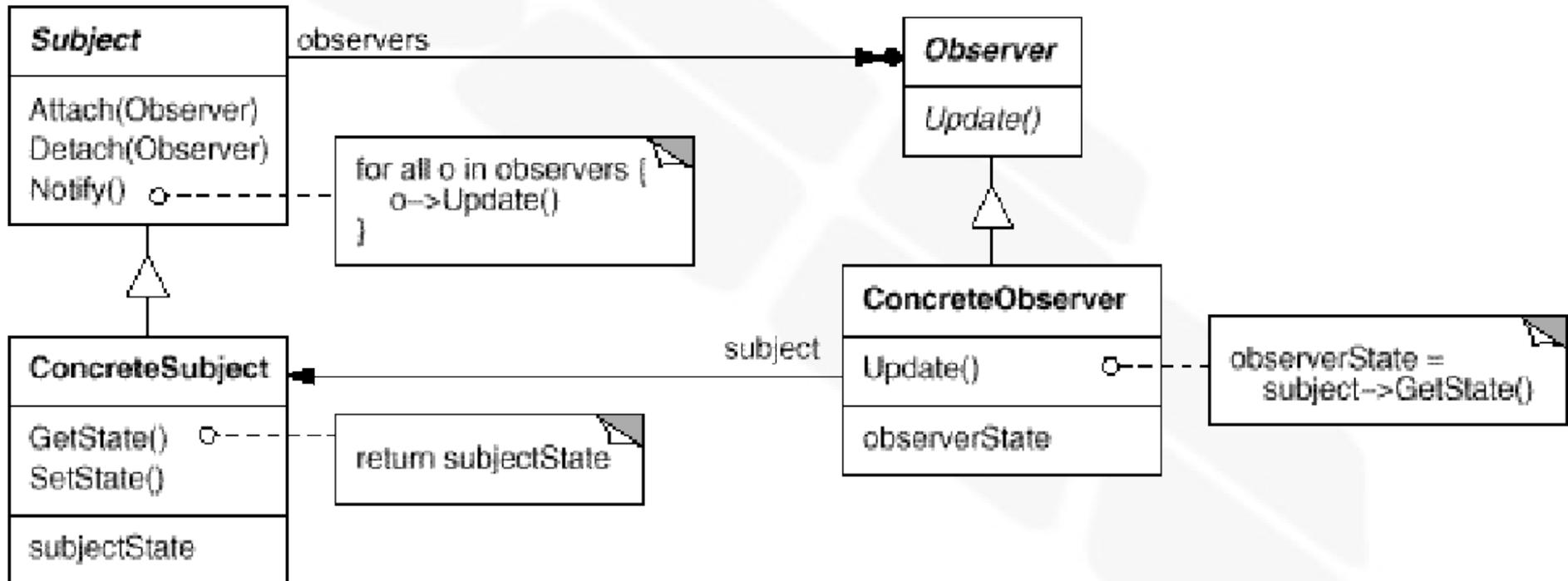
- Objetos chave: *subject* e *observers*
- *Publish-Subscribe*: *observers* são notificados e consultam o(s) *subject(s)* por novas informações

Observer

- Aplicabilidade:
 - Quando uma abstração possui dois aspectos, um dependente do outro. Encapsular estes aspectos em objetos diferentes permite que você os modifique e reutilize de forma independente
 - Quando uma mudança em um objeto requer a mudança em outros e você não sabe quantos e quais objetos precisam ser modificados. Ou seja, não deseja-se um alto acoplamento entre estes objetos

Observer

- Estrutura:



Observer

- Participantes:
 - *Subject*:
 - Conhece seus *observers*. Qualquer número de *observers* podem observar um *subject*
 - Disponibiliza uma interface para anexar e desanexar objetos *observers*
 - *Observer*:
 - Define uma interface de atualização para os objetos que devem ser notificados sobre mudanças no *subject*

Observer

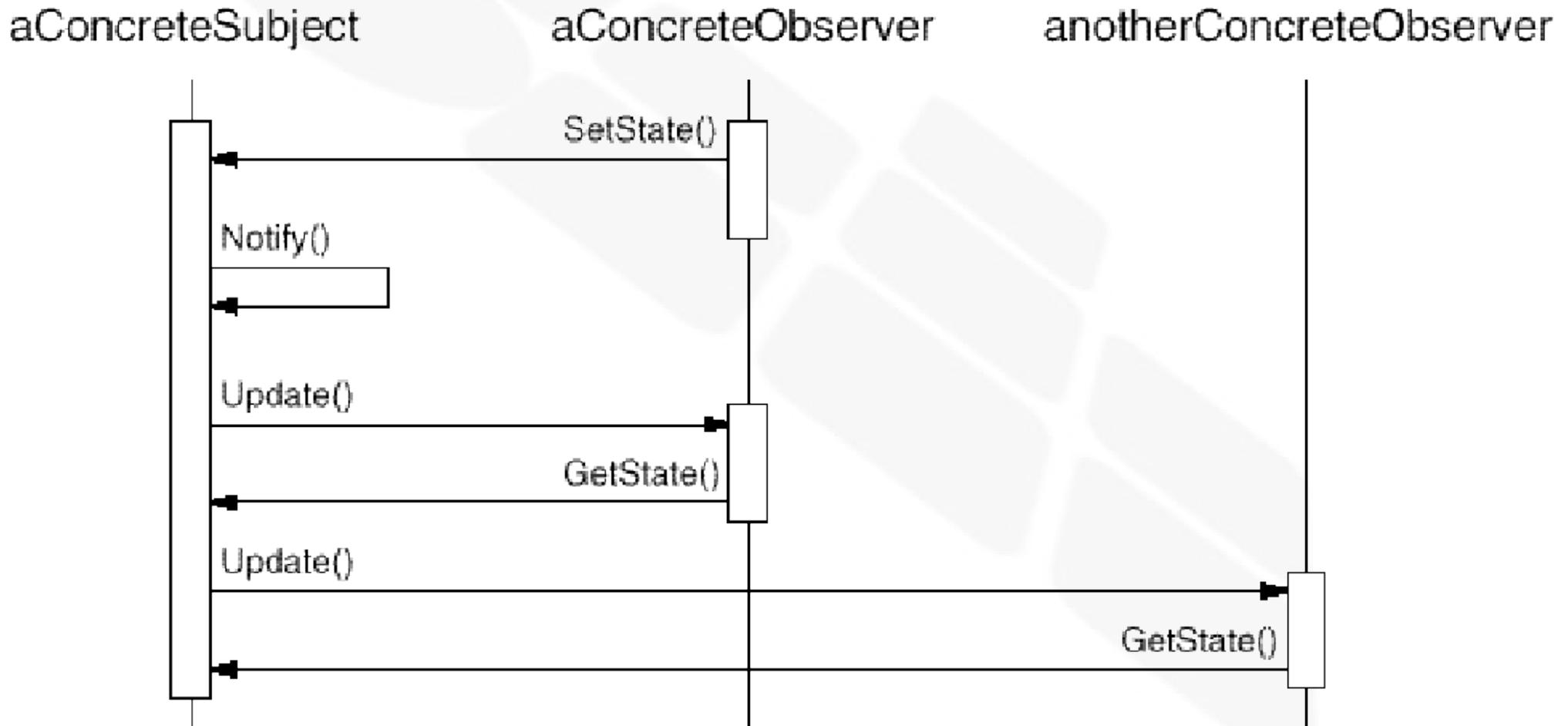
- Participantes:
 - *ConcreteSubject*:
 - Armazena o estado por qual os *ConcreteObservers* têm interesse
 - Envia uma notificação aos seus *observers* sempre que este estado muda
 - *ConcreteObserver*:
 - Mantém referência(s) para o(s) *ConcreteSubject(s)* sendo observado(s)
 - Armazena o estado que deve ser mantido consistente com o estado do *subject*
 - Implementa a interface de atualização do *Observer*

Observer

- Colaborações:
 - O *ConcreteSubject* notifica seus *observers* sempre que uma mudança (que pode tornar o estado do *observer* inconsistente) ocorrer
 - Depois de informado sobre uma mudança no *ConcreteSubject* o *ConcreteObserver* pode consultar o *ConcreteSubject* e utilizar a informação obtida para reconciliar seu estado com o estado do *ConcreteSubject*

Observer

- Colaborações:



Observer

- Conseqüências:
 - Acoplamento abstrato entre o *Subject* e *Observer*:
 - Tudo o que o *subject* sabe é que ele possui uma lista de *observers*, em conformidade com uma interface simples de notificação
 - Pelo fato de serem fracamente acoplados, o *subject* e o *observer* podem pertencer a diferentes camadas de abstração
 - Suporte para comunicação em *broadcast*:
 - Diferente de uma requisição ordinária, o emissor não precisa especificar o receptor e não conhece o número de receptores (flexibilidade em *run-time*)

Observer

- Conseqüências:
 - Atualizações inesperadas:
 - Devido ao fato que *observers* não conhecem a presença de outros *observers* eles não têm ciência do custo final de atualização de um *subject*
 - Operações aparentemente inofensivas no *subject* pode desencadear uma cascata de atualizações
 - Critérios de dependência que não são bem definidos e mantidos podem gerar atualizações espúrias e difíceis de serem rastreadas
 - Um protocolo muito simples de atualização (sem parâmetros) pode fazer com que os *observers* tenham que realizar muito trabalho para deduzir o que mudou

Observer

- Implementação:
 - Mapeando *subjects* aos seus *observers*:
 - Solução simples: o *subject* armazena referências explícitas para os *observers*
 - Entretanto, o consumo de memória pode ser alto se existirem muitos *subjects* e poucos *observers*
 - Alternativa: tabela associativa de *look-up*. *Subjects* sem *observers* não gerariam *overhead* de armazenamento, porém o tempo de acesso aos *observers* seria maior

Observer

- Implementação:
 - Observando mais de um *subject*:
 - É necessário estender a interface de atualização para que o *observer* saiba qual *subject* está enviando a notificação
 - O *subject* pode enviar um referência a si mesmo como um parâmetro do método de atualização

Observer

- Implementação:
 - Quem dispara a atualização ?
 - Opção 1: métodos modificadores do *subject* invocam *notify()* após a mudança do estado
 - Vantagem: clientes não precisam se lembrar de notificar
 - Desvantagem: operações consecutivas disparam atualizações consecutivas podendo gerar ineficiências
 - Opção 2: deixar que os clientes invoquem *notify()* no momento oportuno
 - Vantagem: possibilidade de adiamento da atualização para depois da realização de uma série de modificações no *subject*
 - Desvantagem: aumenta a responsabilidade dos clientes e, portanto, a probabilidade de erros

Observer

- Implementação:
 - Referências pendentes a objetos já removidos:
 - Remover um *subject* não deve gerar referências pendentes nos seus *observers*
 - Possível solução: exigir que os *subjects* notifiquem seus *observers* da sua remoção
 - Deletar os *observers* não é uma opção, pois eles podem estar observando outros *subjects*

Observer

- Implementação:
 - Auto-consistência do estado do *subject* antes da notificação:
 - Os *observers* consultam o(s) *subject(s)* para obter seu(s) novo(s) estados(s) durante a notificação e, portanto, estes estados devem já estar estabilizados
 - A auto-consistência do estado é fácil de ser violada:

```
void MySubject::Operation (int newValue) {  
    BaseClassSubject::Operation(newValue);  
    // trigger notification  
    _myInstVar += newValue;  
    // update subclass state (too late!)  
}
```

Observer

- Implementação:
 - Auto-consistência do estado do *subject* antes da notificação:
 - Solução: *template methods*

```
void Text::Cut (TextRange r) {  
    ReplaceRange(r);    // redefined in subclasses  
    Notify();  
}
```

Observer

- Implementação:
 - Evitando protocolos de atualização específicos de *observer*
 - Geralmente o *subject* envia informações adicionais sobre a mudança sob a forma de parâmetros do método *update()*. Tem-se dois extremos nesta abordagem:
 - **Modelo Push:** o *subject* envia informações detalhadas sobre a mudança, independente se o *observer* as utilizará ou não
 - Pode diminuir o potencial de reuso dos *subjects*
 - **Model Pull:** o *subject* envia somente uma notificação mínima (método *update()* sem parâmetros) e os *observers* consultam o *subject* somente a respeito do que eles necessitam
 - Pode ser ineficiente

Observer

- Implementação:
 - Especificando explicitamente as modificações de interesse
 - Melhora a eficiência das atualizações

```
void Subject::Attach(Observer*, Aspect& interest);
```

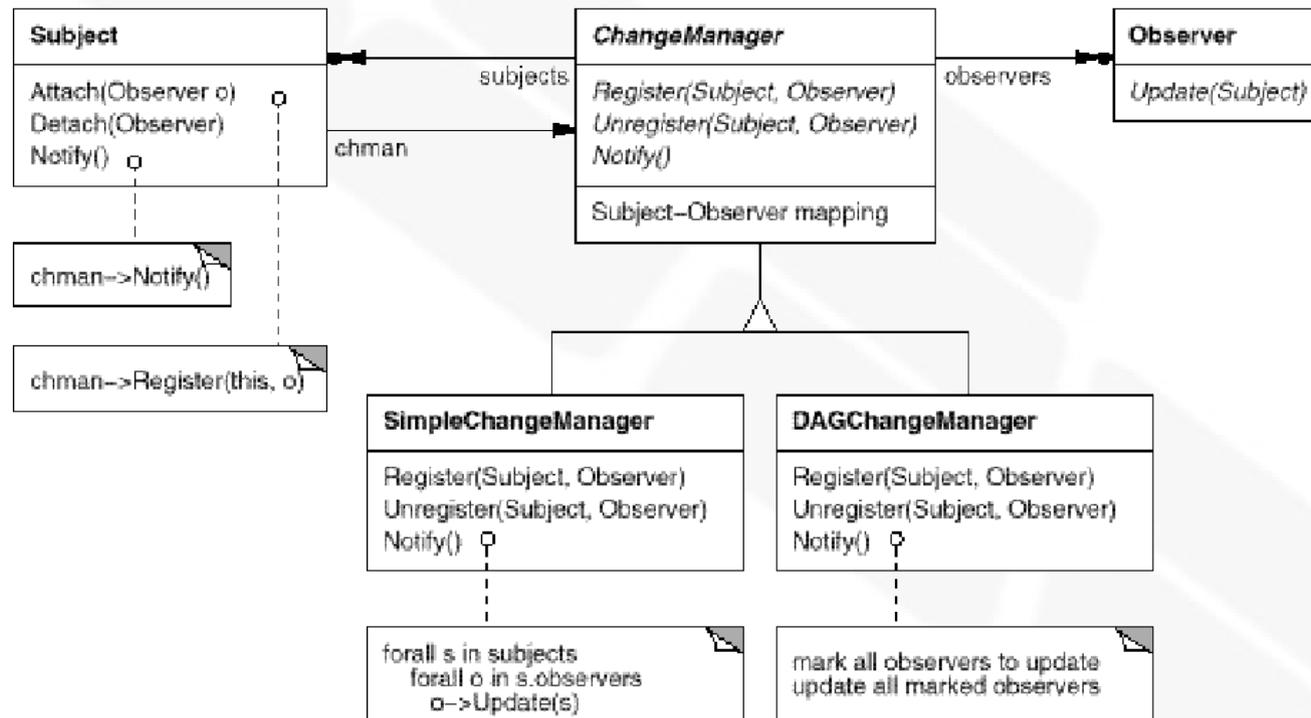
```
void Observer::Update(Subject*, Aspect& interest);
```

Observer

- Implementação:
 - Encapsulando semânticas complexas de atualizações:
 - Neste caso é interessante ter um terceiro objeto para manter tais relações complexas (*ChangeManager*)
 - Responsabilidades do *ChangeManager*:
 - Mapear o *subject* a seus *observers* e disponibilizar uma interface para manter este mapeamento. *Subjects* não mais mantêm referências para os *observers* e vice-versa
 - Definir uma estratégia particular de atualização
 - Atualizar, sempre que solicitado pelo *subject*, todos os *observers* dependentes

Observer

- Implementação:
 - Exemplo do *ChangeManager*:



- Geralmente é também um *mediator* e *singleton*

Observer

- Implementação:
 - Combinando as classes *Subject* e *Observer*
 - Geralmente aplicado em linguagens que não suportam herança múltipla

Observer

- Código exemplo:

```
class Subject;  
  
class Observer {  
public:  
    virtual ~ Observer();  
    virtual void Update(Subject* theChangedSubject) = 0;  
protected:  
    Observer();  
};
```

Observer

- Código exemplo:

```
class Subject {  
public:  
    virtual ~Subject();  
    virtual void Attach(Observer*);  
    virtual void Detach(Observer*);  
    virtual void Notify();  
  
protected:  
    Subject();  
  
private:  
    List<Observer*> *_observers;
```

```
void Subject::Attach (Observer* o) {    _observers->Append(o);    }  
  
void Subject::Detach (Observer* o) {    _observers->Remove(o);    }  
  
void Subject::Notify () {  
ListIterator<Observer*> i(_observers);  
for (i.First(); !i.IsDone(); i.Next()) {  
    i.CurrentItem() ->Update(this);  
}  
}
```

Observer

- Código exemplo:

```
class ClockTimer : public Subject {
public:
    ClockTimer();
    virtual int GetHour();
    virtual int GetMinute();
    virtual int GetSecond();
    void Tick();
};

void ClockTimer::Tick () {
    // update internal time-keeping state
    // ...
    Notify();
}
```

Observer

- Código exemplo:

```
class DigitalClock: public Widget, public Observer {
public:
    DigitalClock(ClockTimer*);
    virtual ~DigitalClock();
    virtual void Update(Subject*);
        // overrides Observer operation
    virtual void Draw();
        // overrides Widget operation;
        // defines how to draw the digital clock
private:
    ClockTimer* _subject;
};
```

Observer

- Código exemplo:

```
DigitalClock::DigitalClock (ClockTimer* s) {
    _subject = s;
    _subject->Attach(this);
}

DigitalClock:: DigitalClock () {
    _subject->Detach(this);
}

void DigitalClock::Update (Subject* theChangedSubject) {
    if (theChangedSubject == _subject) {
        Draw();
    }
}

void DigitalClock::Draw () {
    // get the new values from the subject

    int hour = _subject->GetHour();
    int minute = _subject->GetMinute();
    // etc.

    // draw the digital clock
}
```

Observer

- Código exemplo:

```
class AnalogClock : public Widget, public Observer {  
public:  
    AnalogClock(ClockTimer*);  
    virtual void Update(Subject*);  
    virtual void Draw();  
    // ...  
};
```

```
ClockTimer* timer = new ClockTimer;  
AnalogClock* analogClock = new AnalogClock(timer);  
DigitalClock* digitalClock = new DigitalClock(timer);
```

Observer

- Usos conhecidos:
 - *Smalltalk MVC*
 - ET++
 - THINK
 - *Interviews*
 - *Andrew Toolkit*
 - *Unidraw*

Observer

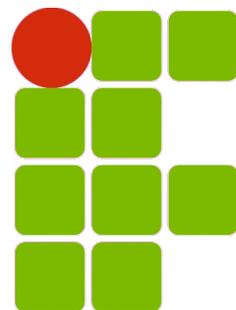
- Padrões relacionados:
 - Ao encapsular semânticas complexas de atualização o *ChangeManager* atua como um *Mediator* entre *subjects* e *observers*
 - O *ChangeManager* pode utilizar o padrão *Singleton* para se tornar acessível de forma única e global

INF011 – Padrões de Projeto

22 – *Observer*

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



**INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA**