PADRÕES DE PROJETO DE SOFTWARE

ACH 2003 — COMPUTAÇÃO ORIENTADA A OBJETOS

Daniel Cordeiro

10 de junho de 2016

Escola de Artes, Ciências e Humanidades | EACH | USP

PADRÕES COMPORTAMENTAIS

PADRÕES COMPORTAMENTAIS

São os padrões relacionados especificamente ao modo como os objetos se comunicam entre si.

- ~~Chain of responsibility~~
- ~~Command~~
- ~~Interpreter~~
- ~~Iterator~~
- ~~Mediator~~
- ~~Memento~~
- Null Object
- Observer
- State
- Strategy
- Template method
- Visitor

NULL OBJECT

OBJETIVO

- Criar uma abstração para a **ausência** de um objeto
- Prover um substituto que ofereça um conjunto de comportamentos padrões para um objeto que não deve fazer nada
- Use Null Object quando:
  - alguma instância de um objeto colaborador não deve fazer nada
  - você quiser abstrair o tratamento de `null` do cliente
  - um objeto requer um colaborador, o Null Objet não introduzirá uma nova colaboração mas usará uma já existente

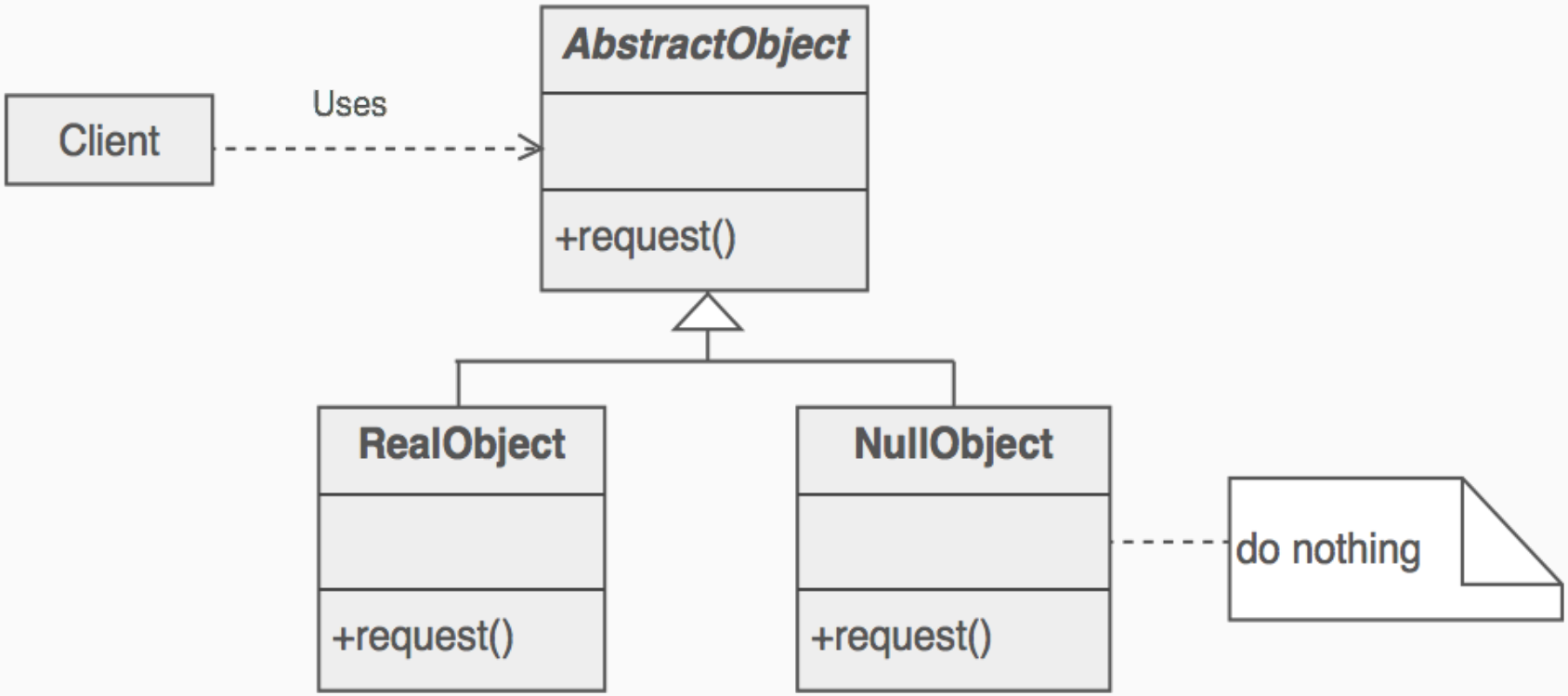
Problema

Dado um objeto que potencialmente pode ser `null` e cujo código que verifica se a referência é `null` não faz nada, ou então usa um valor padrão; como fazer para que a ausência desse objeto (referência `null`) seja tratada de forma transparente?

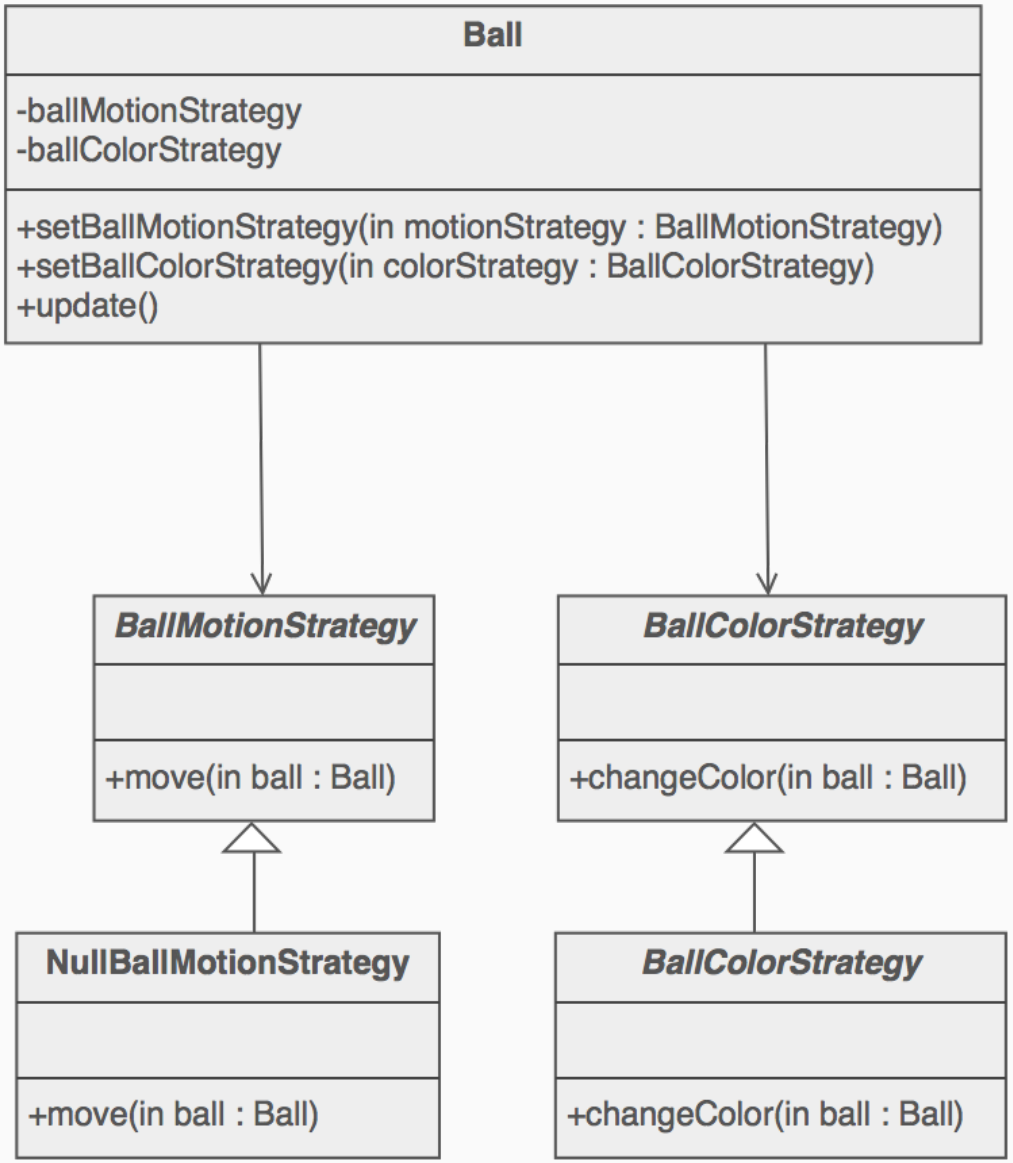
DISCUSSÃO

- Algumas vezes uma classe precisa que um colaborador não faça nada, mas gostaria de tratar esse objeto da mesma forma como trata os que fazem alguma coisa
- Pode ser útil também para a criação de um sistema a partir de suas funcionalidades mais básicas, a mais básica sendo “não fazer nada”
- Null Object pode parecer simples e “estúpido”, mas na verdade um Null Object sempre sabe o que precisa ser feito sem a necessidade de interagir com outros objetos. Isso pode levar a projetos muito “espertos”

ESTRUTURA


AbstractObject
Client
Uses
+request()
RealObject
+request()
NullObject
+request()
do nothing

EXEMPLO


Ball
-ballMotionStrategy
-ballColorStrategy
+setBallMotionStrategy(in motionStrategy : BallMotionStrategy)
+setBallColorStrategy(in colorStrategy : BallColorStrategy)
+update()
BallMotionStrategy
+move(in ball : Ball)
BallColorStrategy
+changeColor(in ball : Ball)
NullBallMotionStrategy
+move(in ball : Ball)
BallColorStrategy
+changeColor(in ball : Ball)

IMPLEMENTAÇÃO

```
class NullOutputStream extends OutputStream {
        public void write(int b) {
                // Não faz nada
        }
}

class NullPrintStream extends PrintStream {
        public NullPrintStream() {
                super(new NullOutputStream());
        }
}

class Application {
        private PrintStream debugout;
        public Application(PrintStream debugout) {
                this.debugout = debugout;
        }

        public void go() {
                int sum = 0;
                for (int i = 0; i < 10; i++) {
                        sum += i;
                        debugout.println("i = " + i);
                }
                System.out.println("sum = " + sum);
        }}
```

OBSERVER

OBJETIVO

- Definir uma dependência um–para–muitos entre objetos de forma que quando um objeto mudar de estado, seus dependentes serão notificados e atualizados automaticamente
- Encapsular os componentes principais do sistema em uma abstração “Assunto” e os componentes variáveis em uma hierarquia de “Observadores”
- A “Visão” no padrão Modelo–Visão–Controlador

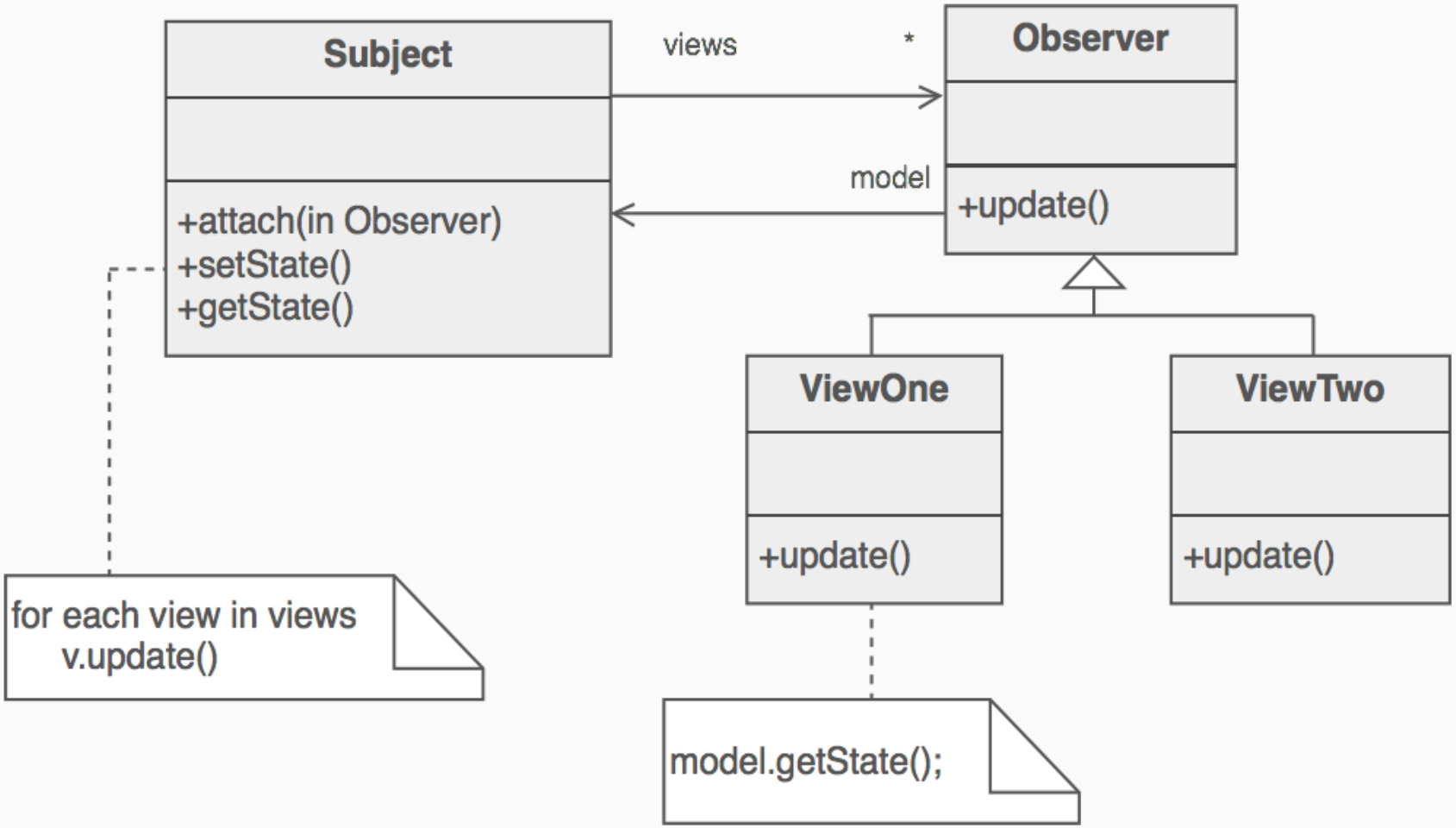
Problema

Um grande projeto monolítico não escala bem quando novos requisitos gráficos ou de monitoração são necessários.

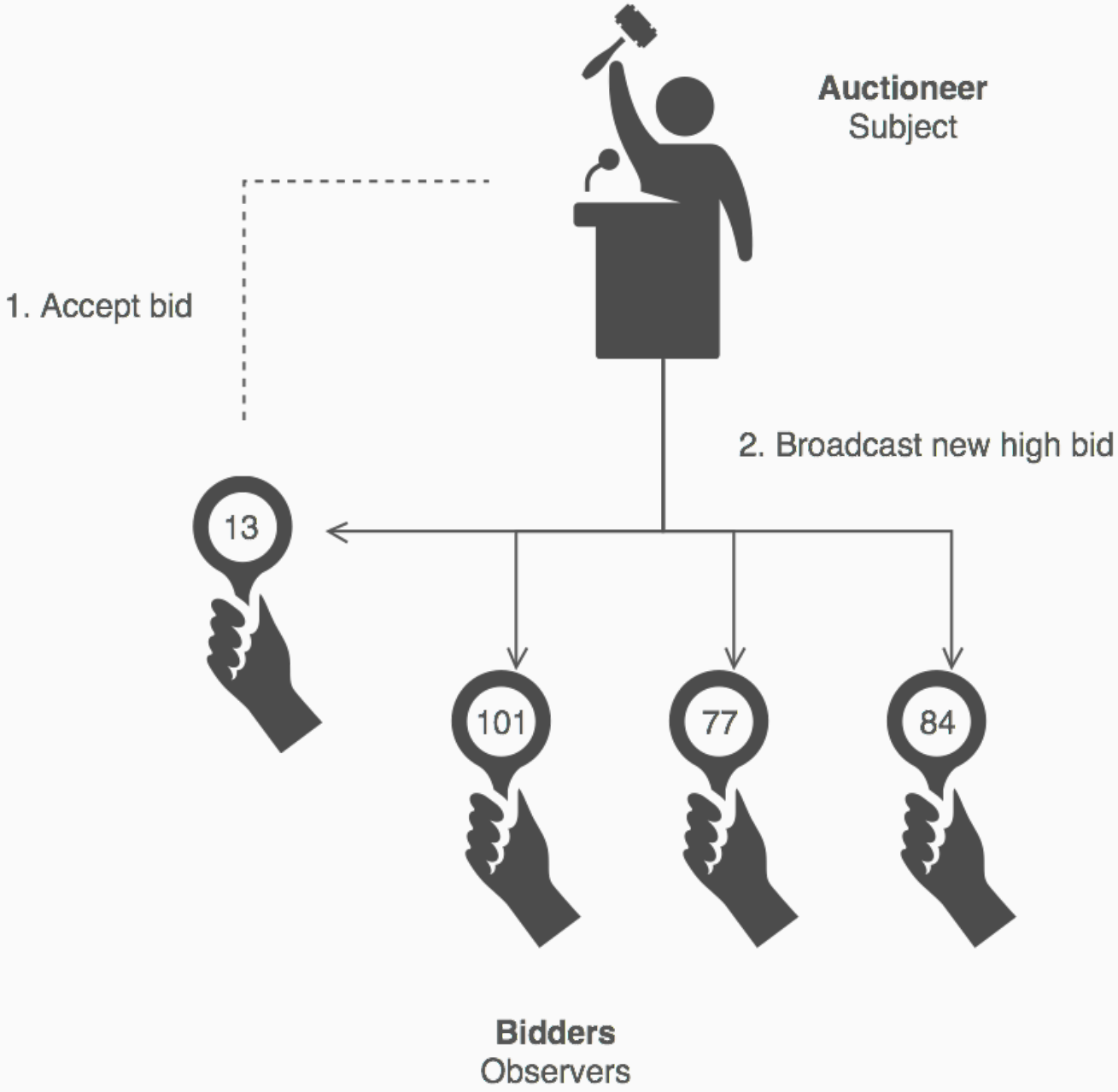
DISCUSSÃO

- Defina um objeto como sendo o “guardião” do modelo de dados ou lógica de negócio (o Assunto)
- Delegue todas as atividades de visualização para objetos Observadores diferentes
- Observadores se registram no Assunto para receber as atualizações sobre o Assunto
- Quando houver mudanças, o Assunto irá notificar todos os Observadores que algo mudou e cada Observador irá consultar o subconjunto do estado do Assunto que ele é responsável por monitorar
- Modelo de interação “pull”. Ao invés do Assunto empurrar (*push*) o que foi modificado goela abaixo dos Observadores, são os Observadores que puxam (*pull*) do Assunto aquilo que eles estão interessados

ESTRUTURA


Subject
views
*
Observer
model
+attach(in Observer)
+setState()
+getState()
+update()
ViewOne
ViewTwo
+update()
+update()
for each view in views
v.update()
model.getState();

EXEMPLO


Auctioneer
Subject
1. Accept bid
2. Broadcast new high bid
13
101
77
84
Bidders
Observers

IMPLEMENTAÇÃO I

```
abstract class Observer {
protected Subject subj;
  public abstract void update();
}

class HexObserver extends Observer {
  public HexObserver( Subject s ) {
    subj = s;
    subj.attach( this );
  }

  public void update() {
  System.out.print( " " + Integer.toHexString( subj.getState() ) );
  }
} // Observadores fazem "pull" da informação
```

IMPLEMENTAÇÃO II

```
class OctObserver extends Observer {
  public OctObserver( Subject s ) {
    subj = s;
    subj.attach( this );
  }
  public void update() {
    System.out.print( " " + Integer.toOctalString( subj.getState() ) );
  }
} // Observadores fazem "pull" da informação



class BinObserver extends Observer {
  public BinObserver( Subject s ) {
    subj = s;
    subj.attach( this ); } // Observador se registra ao assunto
    public void update() {
    System.out.print( " " + Integer.toBinaryString( subj.getState() ) );
  }
}
```

IMPLEMENTAÇÃO III

```
class Subject {
  private Observer[] observers = new Observer[9];
  private int totalObs = 0;
  private int state;
  public void attach( Observer o ) {
    observers[totalObs++] = o;
  }

  public int getState() {
    return state;
  }

  public void setState( int in ) {
    state = in;
    notify();
  }

  private void notify() {
    for (int i=0; i < totalObs; i++) {
      observers[i].update();
    }
  }}
```

Implementação IV

```
public class ObserverDemo {
  public static void main( String[] args ) {
    Subject sub = new Subject();
    // Cliente configura o número e tipo de Observadores
    new HexObserver( sub );
    new OctObserver( sub );
    new BinObserver( sub );
    Scanner scan = new Scaner();
    while (true) {
      System.out.print( "\nEnter a number: " );
      sub.setState( scan.nextInt() );
    }
  }
}
```

LISTA DE VERIFICAÇÃO

1. Faça a diferenciação entre a funcionalidade núcleo e as opcionais
2. Modele a funcionalidade independente como o “assunto” da abstração
3. Modele a funcionalidade opcional como uma hierarquia de “observadores”
4. O Assunto é acoplado apenas à classe base dos Observadores
5. O cliente configura o número e tipo de Observadores
6. Observadores registram-se a um Assunto
7. O Assunto pode “empurrar” (*push*) informação para os Observadores ou os Observadores podem “puxar” (*pull*) as informações de que precisam do Assunto

GIT

- Falamos um pouco sobre o Git, um sistema de controle de versões distribuído
- O Git foi desenvolvido por Linus Torvalds para gerenciar as versões do código fonte do sistema operacional Linux
- Há uma introdução bem sucinta sobre seu uso em `http://rogerdudler.github.io/git-guide/index.pt_BR.html`
- Veja `https://git-scm.com/` e `https://git-scm.com/book/` para uma introdução mais completa
- Aos poucos incorpore o uso de controle de versões no dia a dia do desenvolvimento de seus projetos. Vale a pena!

BIBLIOGRAFIA

- The Gang of Four Book, ou GoF: E. Gamma and R. Helm and R. Johnson and J. Vlissides. Design Patterns — Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
- Alexander Shvets. Design patterns explained simply. `https://sourcemaking.com/design_patterns/`