PROGRAMAÇÃO EM C++
PROJETO FINAL

INF 1900

Prof. Dr. Bruno B. P. Cafeo

Institute of Computing
University of Campinas

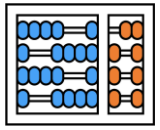
Padrões de Projeto

- O que são?
  - “Descrição de objetos e classes que se comunicam que são adaptados para resolver um problema de projeto geral em um contexto particular”

    [GoF, 1994]

  - “É uma solução particular, que é tão comum quanto efetiva em lidar com um ou mais problemas recorrentes”

    [Fowler et al, 2002]
- Pra que servem?
  - Aplicação similar aos Estilos Arquiteturais:
    - Entretanto, em contextos mais específicos e
    - Em nível mais baixo de abstração



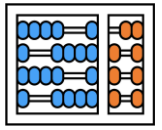
UNICAMP

Padrões de Projeto

Quais são?

GoF (Gang of Four): 23

- Criação
  - ***Abstract Factory***
  - *Builder*
  - *Factory Method*
  - *Prototype*
  - ***Singleton***
- Estruturais
  - ***Adapter***
  - *Bridge*
  - *Composite*
  - *Decorator*
  - ***Façade***
  - *Flyweight*
  - *Proxy*
- Comportamento
  - *Chain of Responsibility*
  - *Command*
  - *Interpreter*
  - *Iterator*
  - *Mediator*
  - *Memento*
  - ***Observer***
  - *State*
  - *Strategy*
  - *Template Method*
  - *Visitor*

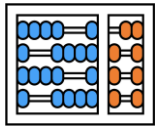


UNICAMP

Padrões de Projeto

Quais são?

- GRASP (*General Responsibility Assignment Software Patterns*)
  - *Creator*
  - *Information Expert*
  - *Controller*
  - *Low Coupling*
  - *High Cohesion*
  - *Polymorphism*
  - *Pure Fabrication*
  - *Indirection*
  - *Protected Variations*
- POSA (Pattern-Oriented Software Architecture)
  - *From Mud To Structure*
  - *Distributed Infrastructure*
  - *Event Demultiplexing and Dispatching*
  - *Interface Partitioning*
  - *Component Partitioning*
  - *Application Control*
  - *Concurrency*
  - *Synchronization*
  - *Object Interaction*
  - *Adaptation and Extension*
  - *Modal Behavior*
  - *Resource Management*
  - *Database Access*



UNICAMP

Padrões de Projeto

Formato

- *Nome*
- *Padrões Relacionados*
- *Contexto*
- *Problema*
- *Solução/Estrutura*
- *Participantes*
- *Pseudocódigo*
- *Consequências*

Pattern name and maturity level
Inbound patterns
Context
Border line
Problem statement
Forces
Solution instruction
Solution sketch
A role
An interaction between roles
Numbers indicate ordering of steps.
An interface
Pseudo-code to illustrate role-internals and role behavior
Solution structure and behavior
Solution consequences
Solution details and outbound patterns

Reactor **

When developing event-driven software, or designing a CLIENT REQUEST HANDLER (246) or a SERVER REQUEST HANDLER (249) or a SERVER REQUEST HANDLER (249) ...

... we must decouple infrastructure behavior associated with detecting, demultiplexing, and dispatching events from short-duration components that service the events.

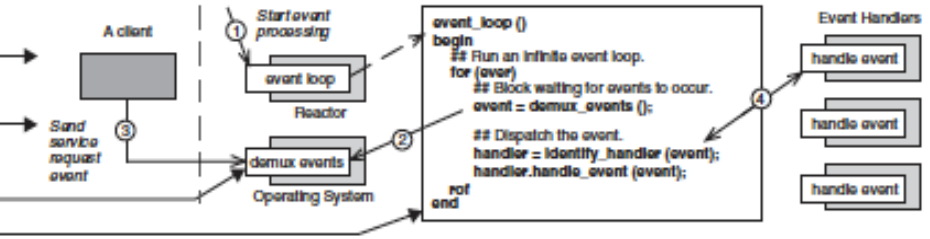
♦ ♦ ♦

Event-driven software often receives service request events from multiple event sources, which it demultiplexes and dispatches to event handlers that perform further service processing. Events can also arrive simultaneously at the event-driven application. However, to simplify development, events should be processed sequentially and synchronously.

Efficiently and flexibly processing events that arrive concurrently from multiple sources is hard. For example, using multi-threading to wait for events to occur in a set of event sources can introduce overheads due to synchronization, context switching, and data movement. In contrast, blocking indefinitely on a single event source can prevent the servicing of other event sources, degrading the quality of service to clients. [...].

Therefore:

Provide an event handling infrastructure that waits on multiple event sources simultaneously for service request events to occur, but only demultiplexes and dispatches one event at a time to a corresponding event handler that performs the service.

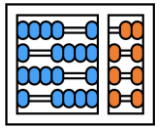


A reactor component coordinates the processing of events within the event-driven application. It defines an event loop that uses an operating system event demultiplexer to wait synchronously for service request events to occur on a set of event sources. By delegating the demultiplexing of events to the operating system, the reactor can wait for multiple event sources simultaneously without multi-threading the application code. [...].

♦ ♦ ♦

There are several benefits of a REACTOR design. First, operating system event demultiplexing mechanisms can wait on a set of event sources while avoiding the performance overhead and programming complexity associated with multi-threading. Second, encapsulating the software event loop within the reactor shields service event handlers from complexities in the synchronous event demultiplexing and dispatching infrastructure. [...].

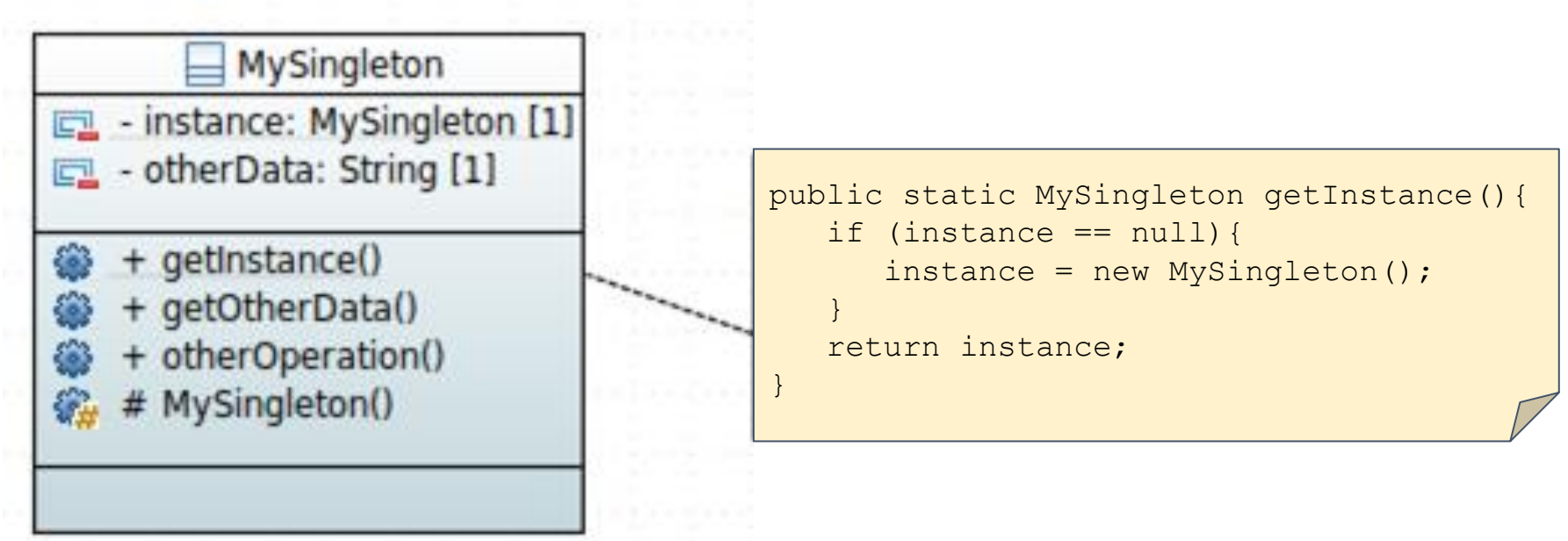
Different reactor implementations are often required when platforms offer different event demultiplexers. In such a situation, an EXPLICIT INTERFACE (281) may be needed to separate the reactor interface from its implementations. Event handlers are often arranged in an ACCEPTOR-CONNECTOR (265) configuration, where service handlers provide domain-specific functionality and acceptors and connectors establish connections on behalf of service event handlers. [...].



Padrões de Projeto

Singleton

Garante que uma classe possua uma única instância, e provê um ponto de acesso global a essa.

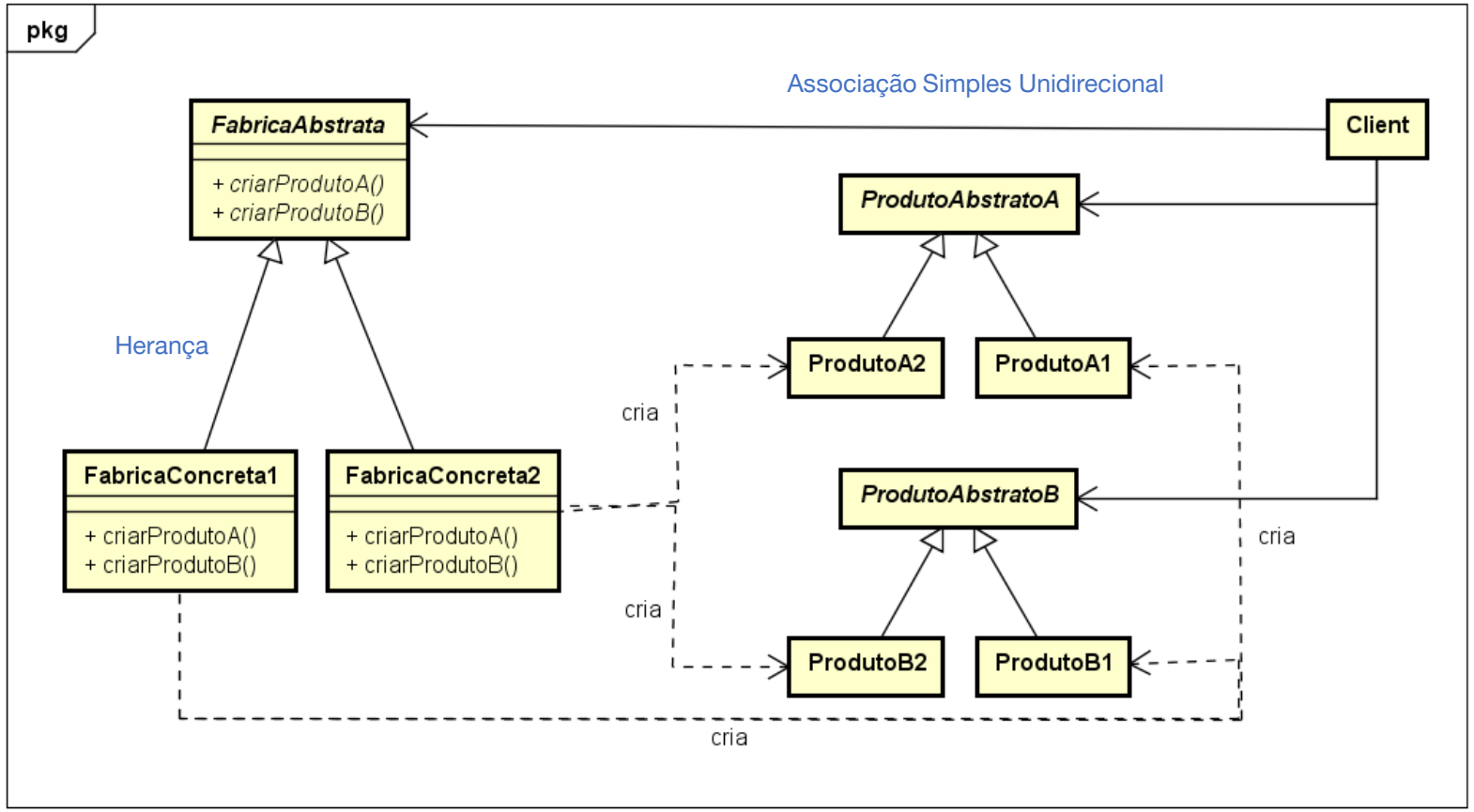


#O construtor real da classe deve ser privado

Padrões de Projeto

Abstract Factory

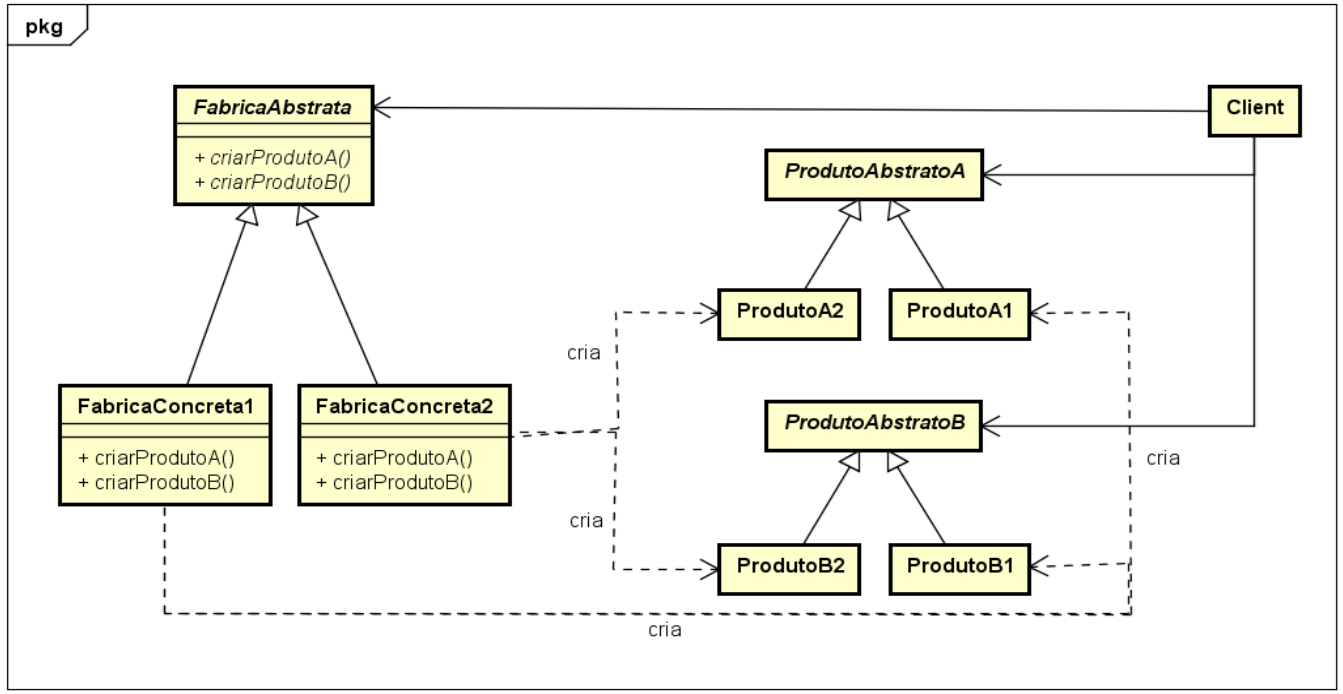
Provê uma classe para criar famílias de objetos relacionados ou dependentes sem especificar suas classes concretas

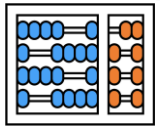


Padrões de Projeto

Abstract Factory

```
public class ExemploFabrica {

 public static void main(String[] args) {

    FabricaAbstrata fabrica1 = new FabricaConcreta1();
    Cliente cliente1 = new Cliente(fabrica1);
    cliente1.executar(); FabricaAbstrata fabrica2 = new
FabricaConcreta2();
    Cliente cliente2 = new Cliente(fabrica2);
    cliente2.executar();
 }
}

class Cliente {
 private ProdutoAbstratoA produtoA;
 private ProdutoAbstratoB produtoB;

 Cliente(FabricaAbstrata fabrica) {
    produtoA = fabrica.createProdutoA();
    produtoB = fabrica.createProdutoB();
 }

 void executar() {
    produtoB.interagir(produtoA);
 }
}
```

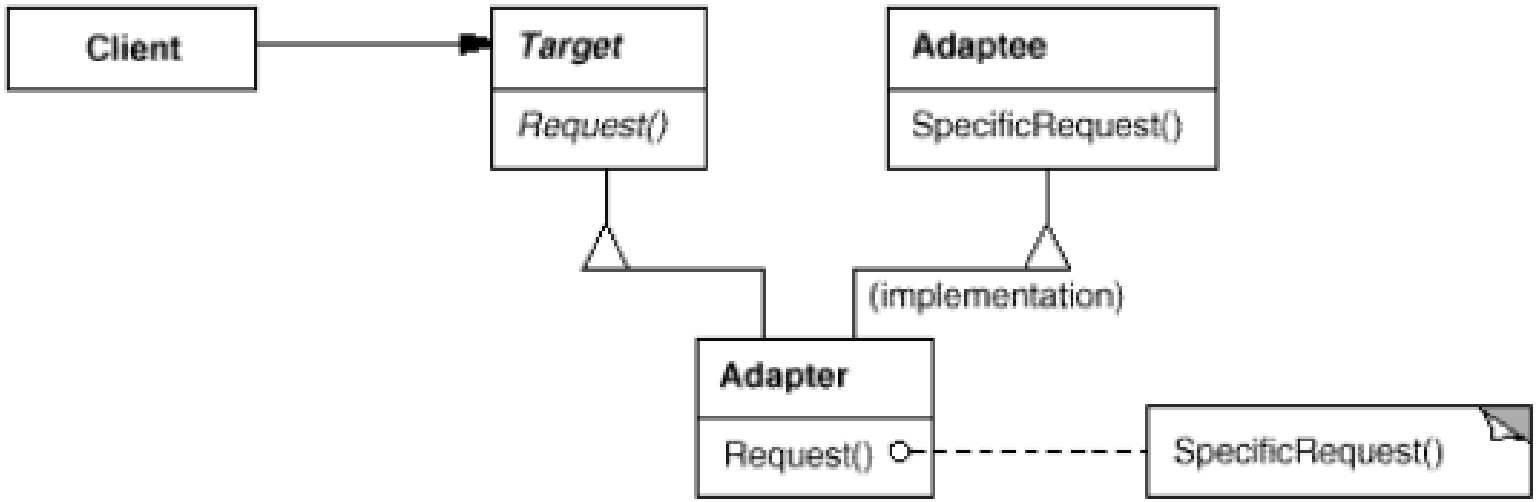




Padrões de Projeto

Adapter (Wrapper)

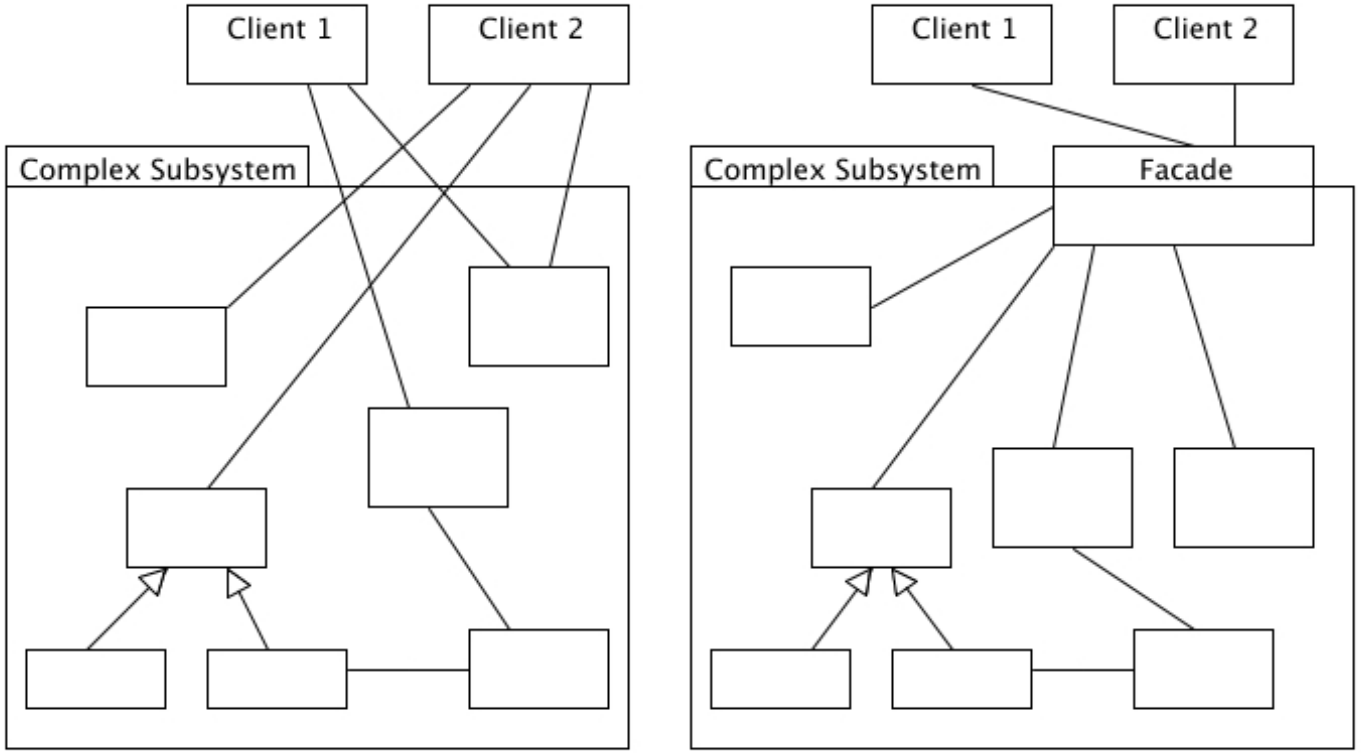
Converte a interface de uma classe em outra interface que o cliente espera. Torna interfaces compatíveis.



Padrões de Projeto

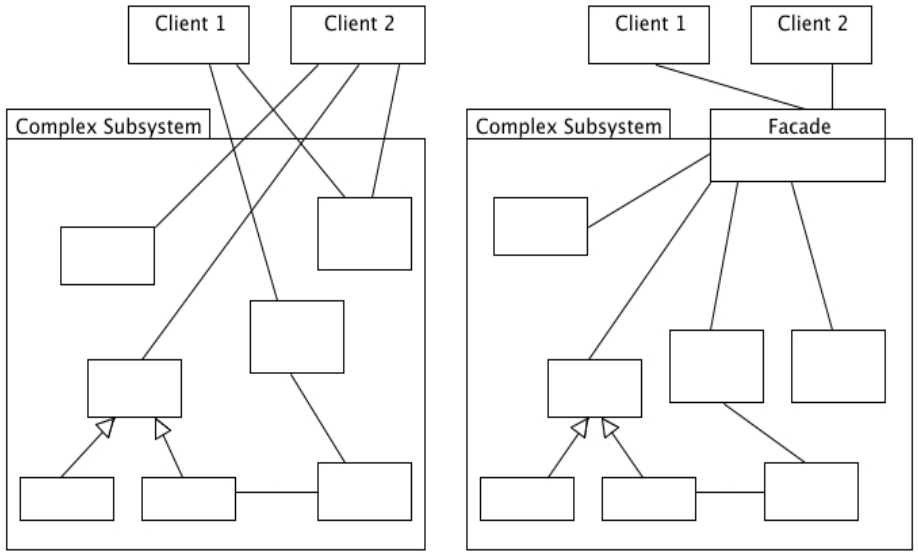
Façade

Provê uma interface unificada para um conjunto de interfaces em um subsistema, simplificando o acesso a este subsistema.



Padrões de Projeto

Façade




```
class ClasseInternaA {

 public bool boolMethod(){…}
 public void doIT(){…}
}

class ClasseInternaB {

 public void doSomething() {…}
 public void doAnotherThing()
{…}
 public void intB.doNothing()
{…}
}
```

```
public class ExemploFachada {

 public static void main(String[] args) {

    Facade fachada = new Facade();
    fachada.executarAcao1();
    fachada.executarAcao2();
 }
}

class Facade {
 private ClasseInternaA intA;
 private ClasseInternaB intB;

 void executarAcao1() {
    if(intA.boolMethod()){
       intB.doSomething();
       intB.doAnotherThing ();
    }
    else{
       intA.doIt();
       intB.doNothing();
    }
 }

 void executarAcao2() {
    intA.doIt();
    intB.doAnotherThing();
 }
}
```

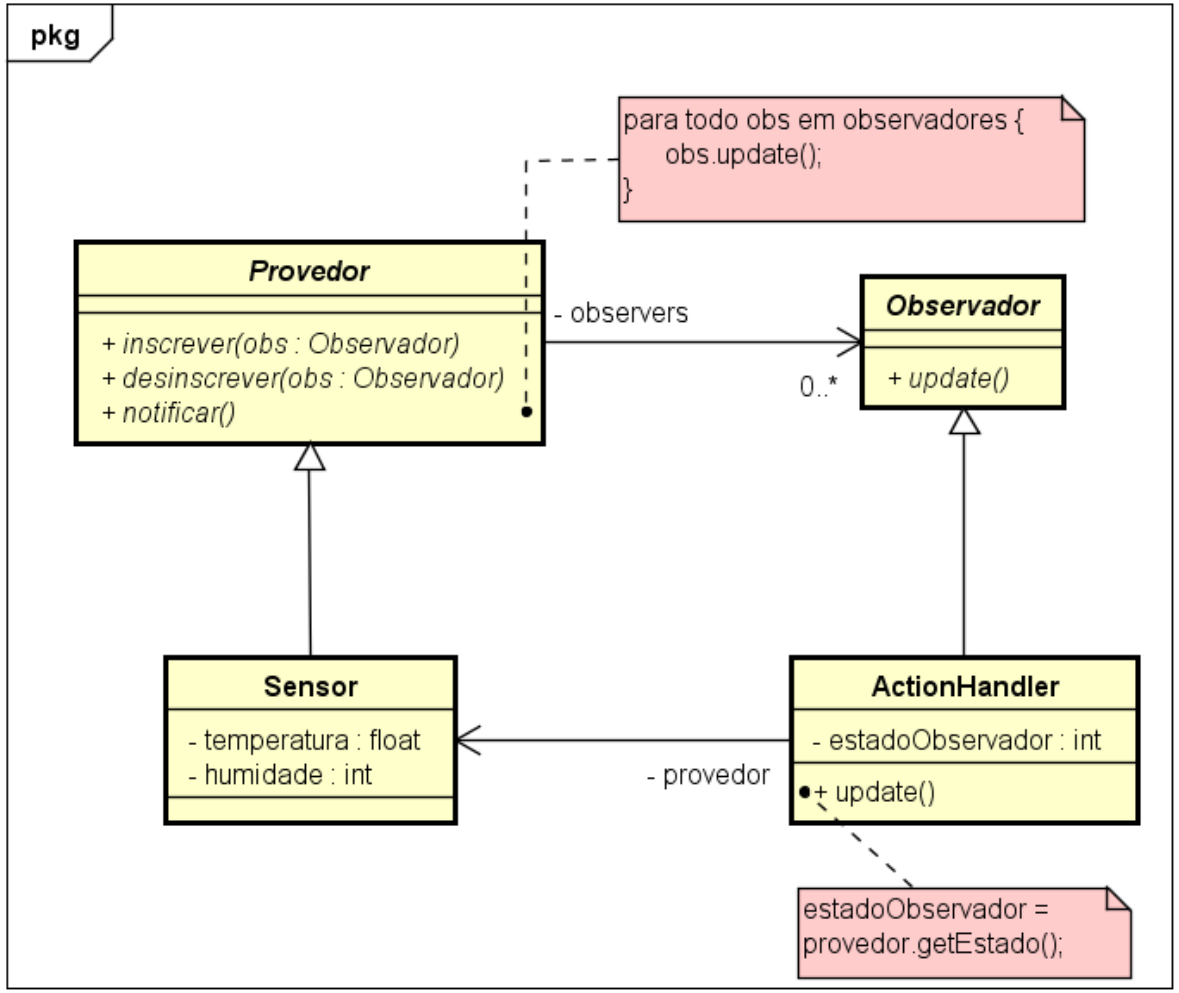

UNICAMP

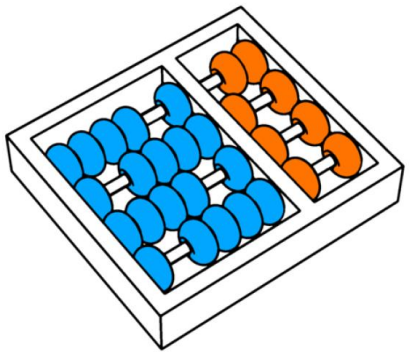
Padrões de Projeto

Observer

Define **dependência** 1:N entre objetos

Quando o estado do objeto muda, todos de que dele dependem são **notificados** e **atualizados** automaticamente





INSTITUTO DE COMPUTAÇÃO

UNICAMP

Prof. Dr. Bruno B. P. Cafeo

Sala 04
Instituto de Computação - Unicamp
Av. Albert Einstein, 1251
Cidade Universitária
Campinas – SP
13083-852

https://ic.unicamp.br/~cafeo/
cafeo@ic.unicamp.br