Padrões de Projeto de Software Orientado a Objetos

Introdução

Profa. Thienne Johnson

Conteúdo

- E. Gamma and R. Helm and R. Johnson and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
  - Conhecido como GoF (Gang of Four)
  - Versão em português disponível na biblioteca da EACH. (nome: Padrões de Projeto)

Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

- Java Design Patterns At a Glance
  - http://www.javacamp.org/designPattern
- Java Design Patterns Resource Center
  - http://www.deitel.com/ResourceCenters/Programming/JavaDesignPatterns/tabid/1103/Default.aspx

O problema

- Projetar software para reúso é difícil porque deve-se procurar por:
  - Uma boa decomposição do problema e a abstração correta.
  - Flexibilidade, modularidade e elegância.

- Projetos freqüentemente emergem de um processo iterativo (tentativas e muitos erros).

O problema

- A boa notícia é que bons projetos existem:
  - na verdade, eles apresentam características recorrentes,
  - mas eles quase nunca são idênticos.

- Perspectiva de engenharia: os projetos podem ser descritos, codificados ou padronizados?
  - isto iria diminuir a fase de tentativa e erro.
  - sofwares melhores seriam produzidos mais rápidos.

Padrões de Projeto de Software OO

- Também conhecidos como
  - *Padrões de Desenho de Software OO*
  - ou simplesmente como *Padrões*.

A Inspiração

- A idéia de padrões foi apresentada por Christopher Alexander em 1977 no contexto de Arquitetura (de prédios e cidades):

Cada padrão descreve um problema que ocorre repetidamente de novo e de novo em nosso ambiente, e então descreve a parte central da solução para aquele problema de uma forma que você pode usar esta solução um milhão de vezes, sem nunca implementa-la duas vezes da mesma forma.

Quantos padrões de projetos existem?

- Muitos. Um site relata ao menos 250 PP. E muito mais são criados a cada dia.

- PP não são idiomas, algortimos ou componentes.

Qual a relação entre esses padrões?

- Para criar um sistema, podem ser necessários vários padrões.

- Diferentes designers podem usar diferentes padrões para resolver o mesmo problema.

- Geralmente:
  - Alguns padrões se ‘encaixam’ naturalmente
  - Um padrão pode levar a outro
  - Alguns padrões são similares e alternativos
  - Padrões podem ser descobertos e documentados
  - Padrões não são métodos ou frameworks
  - Padrões dão uma dica para resolver um problema de forma efetiva

Catálogo de soluções

- Um padrão encerra o conhecimento de uma pessoa muito experiente em um determinado assunto de uma forma que este conhecimento pode ser transmitido para outras pessoas menos experientes.
- Outras ciências (p.ex. química) e engenharias possuem catálogos de soluções.
- Desde 1995, o desenvolvimento de software passou a ter o seu primeiro catálogo de soluções para projeto de software: o livro GoF.

Gang of Four (GoF)

- Passamos a ter um vocabulário comum para conversar sobre projetos de software.
- Soluções que não tinham nome passam a ter nome.
- Ao invés de discutirmos um sistema em termos de pilhas, filas, árvores e listas ligadas, passamos a falar de coisas de muito mais alto nível como Fábricas, Fachadas, Observador, Estratégia, etc.

Gang of Four (GoF)

- A maioria dos autores eram entusiastas de Smalltalk, principalmente o Ralph Johnson.
- Mas acabaram baseando o livro em C++ para que o impacto junto à comunidade de CC fosse maior. E o impacto foi enorme, o livro vendeu centenas de milhares de cópias.

O Formato de um padrão

- Todo padrão inclui
  - Nome
  - Problema
  - Solução
  - Conseqüências / Forças

- Existem outros tipos de padrões mas vamos nos concentrar no GoF.

O Formato dos padrões no GoF

1. Nome (inclui número da página)
   - um bom nome é essencial para que o padrão caia na boca do povo
2. Objetivo / Intenção
3. *Também Conhecido Como*
4. Motivação
   - um cenário mostrando o problema e a necessidade da solução
5. Aplicabilidade
   - como reconhecer as situações nas quais o padrão é aplicável

O Formato dos padrões no GoF

6. Estrutura
   - uma representação gráfica da estrutura de classes do padrão
     (usando OMT91) em, às vezes, diagramas de interação (Booch 94)
7. Participantes
   - as classes e objetos que participam e quais são suas responsabilidades
8. Colaborações
   - como os participantes colaboram para exercer as suas responsabilidades

O Formato dos padrões no GoF

9. Conseqüências
   - vantagens e desvantagens, *trade-offs*
10. Implementação
    - com quais detalhes devemos nos preocupar quando implementamos o padrão
    - aspectos específicos de cada linguagem
11. Exemplo de Código
    - no caso do GoF, em C++ (a maioria) ou Smalltalk

O Formato dos padrões no GoF

12. Usos Conhecidos
    - exemplos de sistemas reais de domínios diferentes onde o padrão é utilizado
13. Padrões Relacionados
    - quais outros padrões devem ser usados em conjunto com esse
    - quais padrões são similares a este, quais são as dierenças

Tipos de Padrões de Projeto

- Categorias de Padrões do GoF
  - Padrões de Criação
  - Padrões Estruturais
  - Padrões Comportamentais

Os 23 Padrões do GoF
Criação

- Processo de criação de Objetos
  - Abstract Factory
  - Builder
  - Factory Method
  - Prototype
  - Singleton

Os 23 Padrões do GoF
Estruturais

- Composição de classes ou objetos
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Façade
  - Flyweight
  - Proxy

Os 23 Padrões do GoF
Comportamentais

• Forma na qual classes ou objetos interagem e distribuem responsabilidades

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

Organizando o catálogo

Escopo		Propósito		
		Criação	Estrutural	Comportamental
	Classe	Factory Method (121)	Adapter (157)	Interpreter (274) Template Method (360)
	Objeto	Abstract Factory (99) Builder (110) Prototype (133) Singleton (144)	Adapter (157) Bridge (171) Composite (183) Decorator (196) Facade (208) Flyweight (218) Proxy (233)	Chain of Responsibility (251) Command (263) Iterator (289) Mediator (305) Memento (316) Observer (326) State (338) Strategy (349) Visitor (366)

•Escopo‘Classe’ negociam relaçionamentos entre classes e sub-classes. Essas relações são estabelecidas através de herança. Então são estáticas – fixas em tempo de compilação.

•Escopo ‘Objeto’ negociam com relações de objetos, que podem mudar em tempo de execução.

Relação entre padrões

Memento
Proxy
Adapter
Builder
saving state of iteration
Iterator
avoiding hysteresis
Bridge
creating composites
enumerating children
adding responsibilities to objects
composed using
Command
Composite
Decorator
sharing composites
adding operations
defining traversals
defining the chain
Flyweight
defining grammar
Visitor
changing skin versus guts
sharing strategies
Interpreter
adding operations
Chain of Responsibility
Strategy
sharing states
sharing terminal symbols
Mediator
complex dependency management
Observer
State
defining algorithm's steps
Template Method
often uses
Prototype
Factory Method
configure factory dynamically
implement using
Abstract Factory
single instance
Facade
single instance
Singleton

Por onde começar?

Ordem sugerida

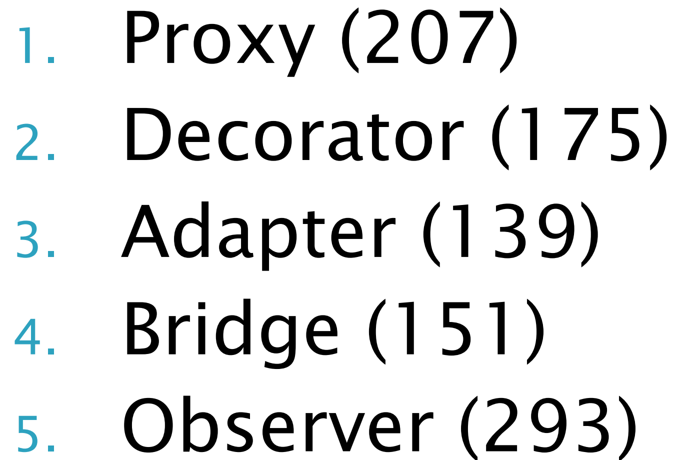
- http://mahemoff.com/paper/software/learningGoFPatterns/

- Fácil
- Intermediário
- Avançado

Fácil

1. Facade (185)
2. Singleton (127)
3. Mediator (273)
4. Iterator (257)
5. Strategy (315)
6. Command (233)
7. Builder (97)
8. State (305)
9. Template Method (325)
10. Factory Method (107)
11. Memento (283)
12. Prototype (117)

Intermediários



1. Proxy (207)
2. Decorator (175)
3. Adapter (139)
4. Bridge (151)
5. Observer (293)

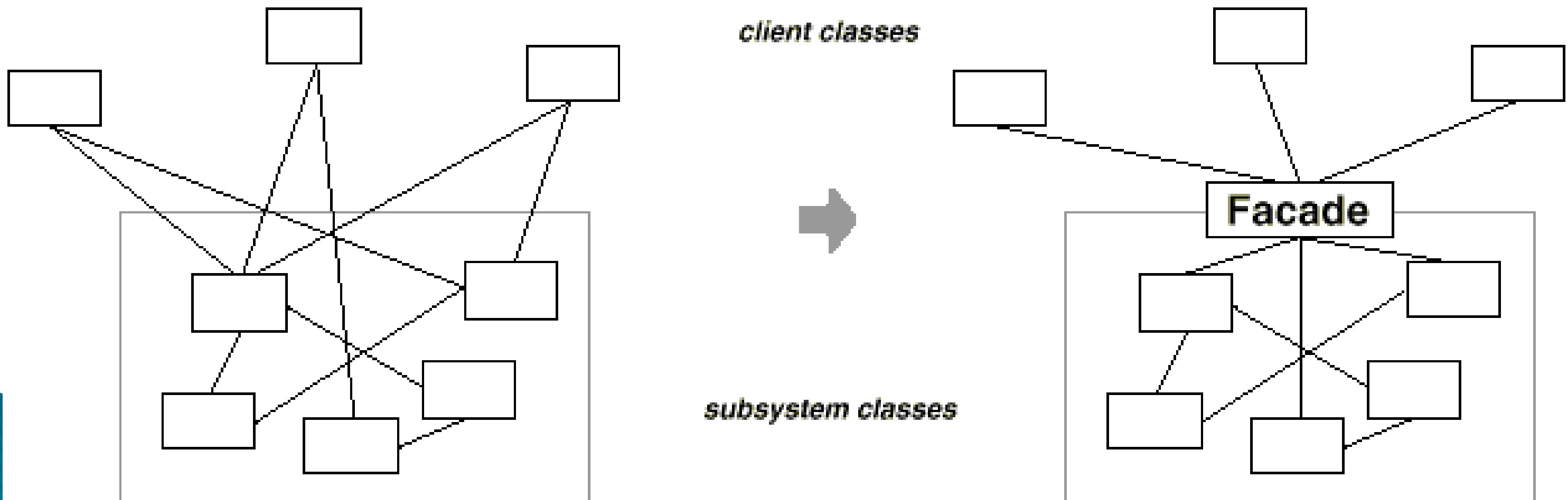
Avançados

1. Composite (163)
2. Interpreter (243)
3. Chain Of Responsibility (223)
4. Abstract Factory (87)
5. Visitor (331)

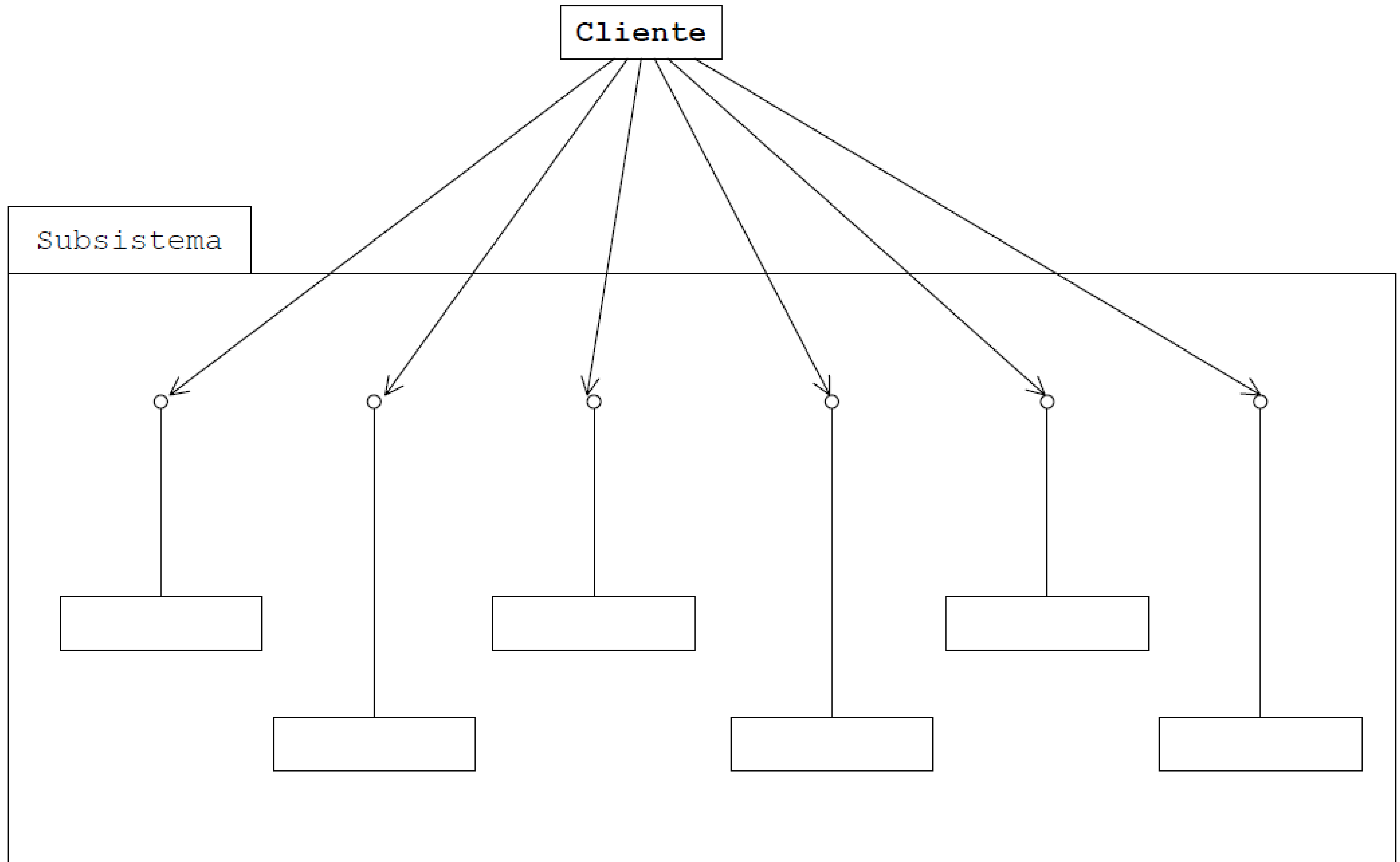
Facade (Façade)

Facade (*Fachada*)

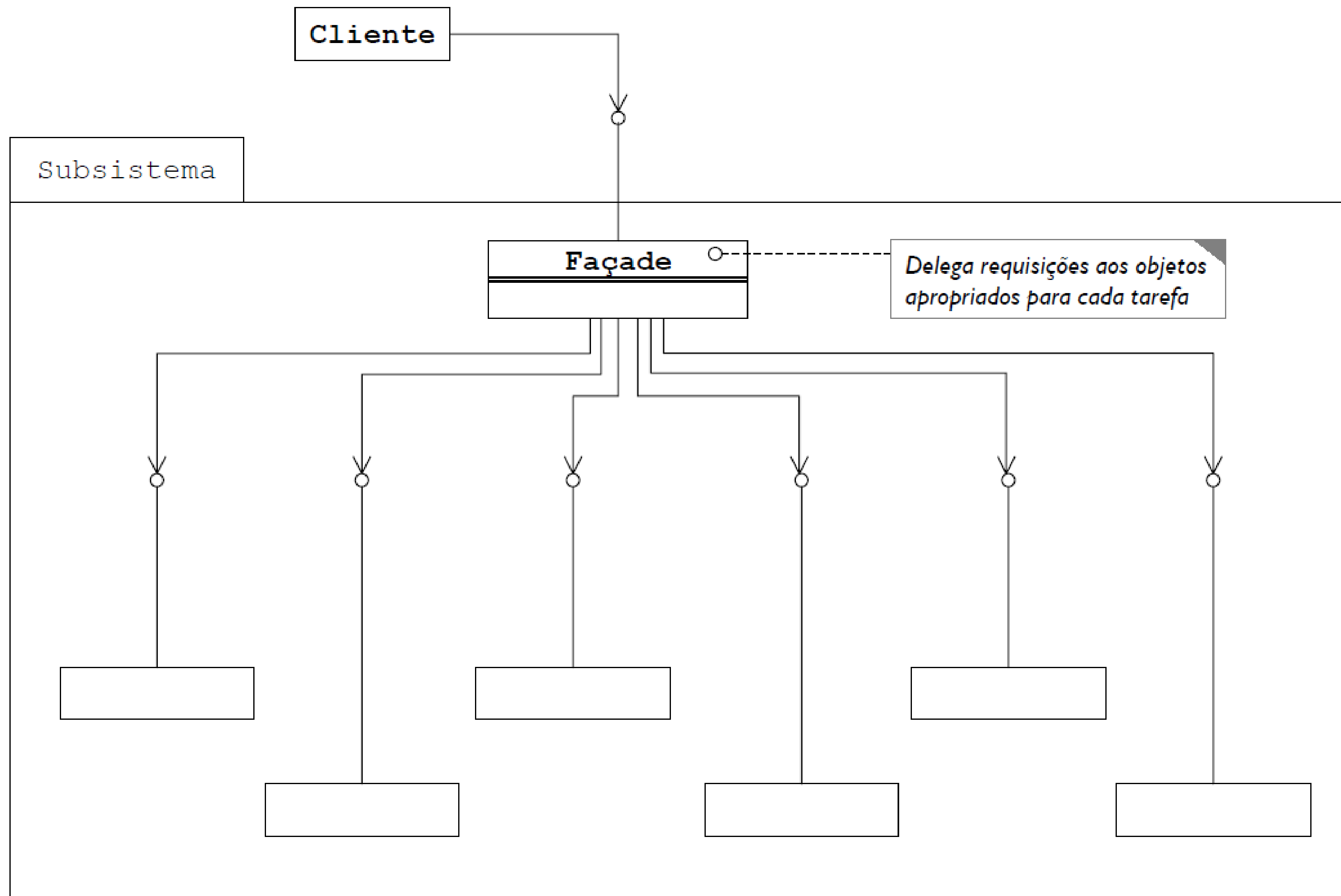
- Provê uma interface unificada para um conjunto de interfaces em um subsistema.
- Define uma interface de mais alto nível que torna mais fácil o uso do subsistema.



Antes



Usando Façade



Aplicações

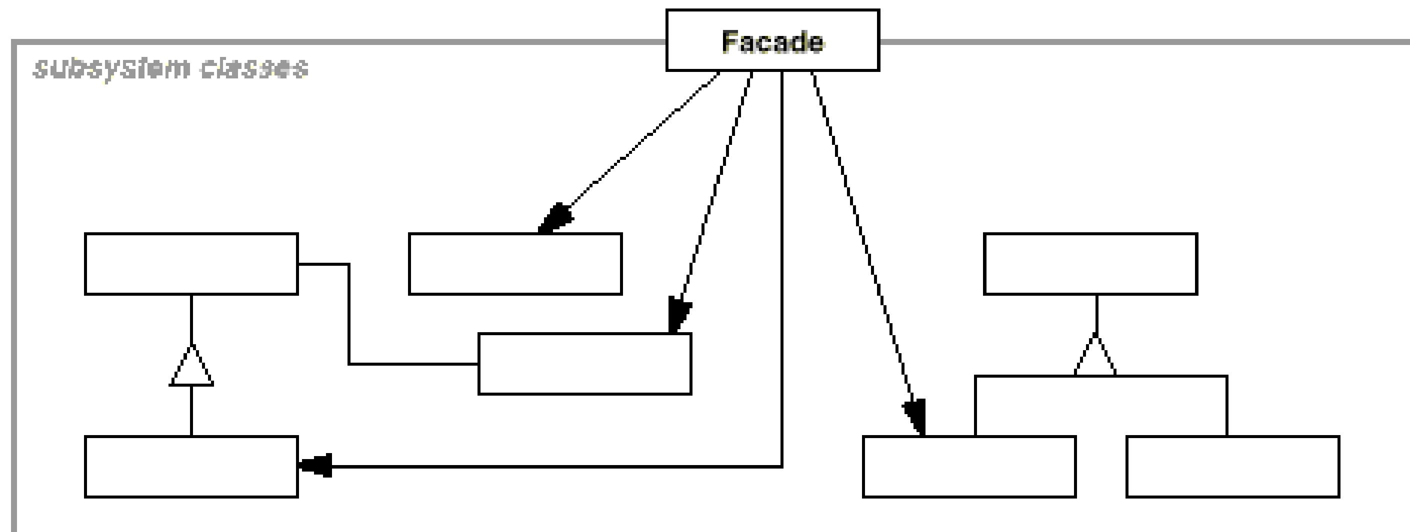
- Oferece uma interface simples para um subsistema complexo

- Existem muitas dependências entre clientes e as classes de implementação de uma abstração.
  - A introdução de um "Façade" irá desacoplar o subsistema dos clientes dos outros subsistemas, promovendo assim, a independência e portabilidade desses subsistemas.

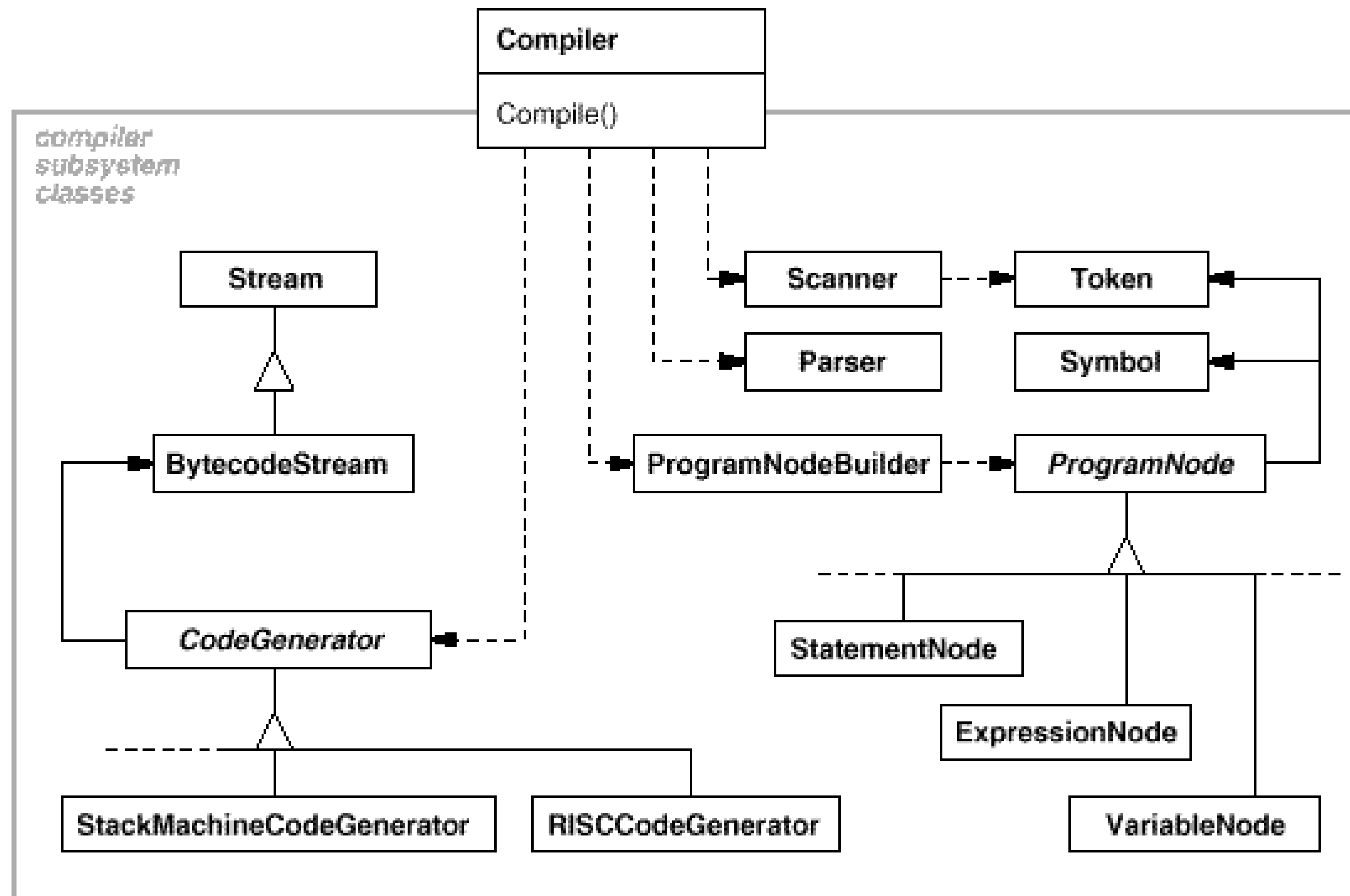
Aplicações

- Quando se deseja subsistemas em camadas.
- Quando se quer definir um ponto de entrada para cada nível do subsistema.
  - Se os subsistemas são dependentes, então pode-se simplificar a dependência entre eles fazendo com que eles se comuniquem uns com os outros unicamente através dos seus "Façades“.

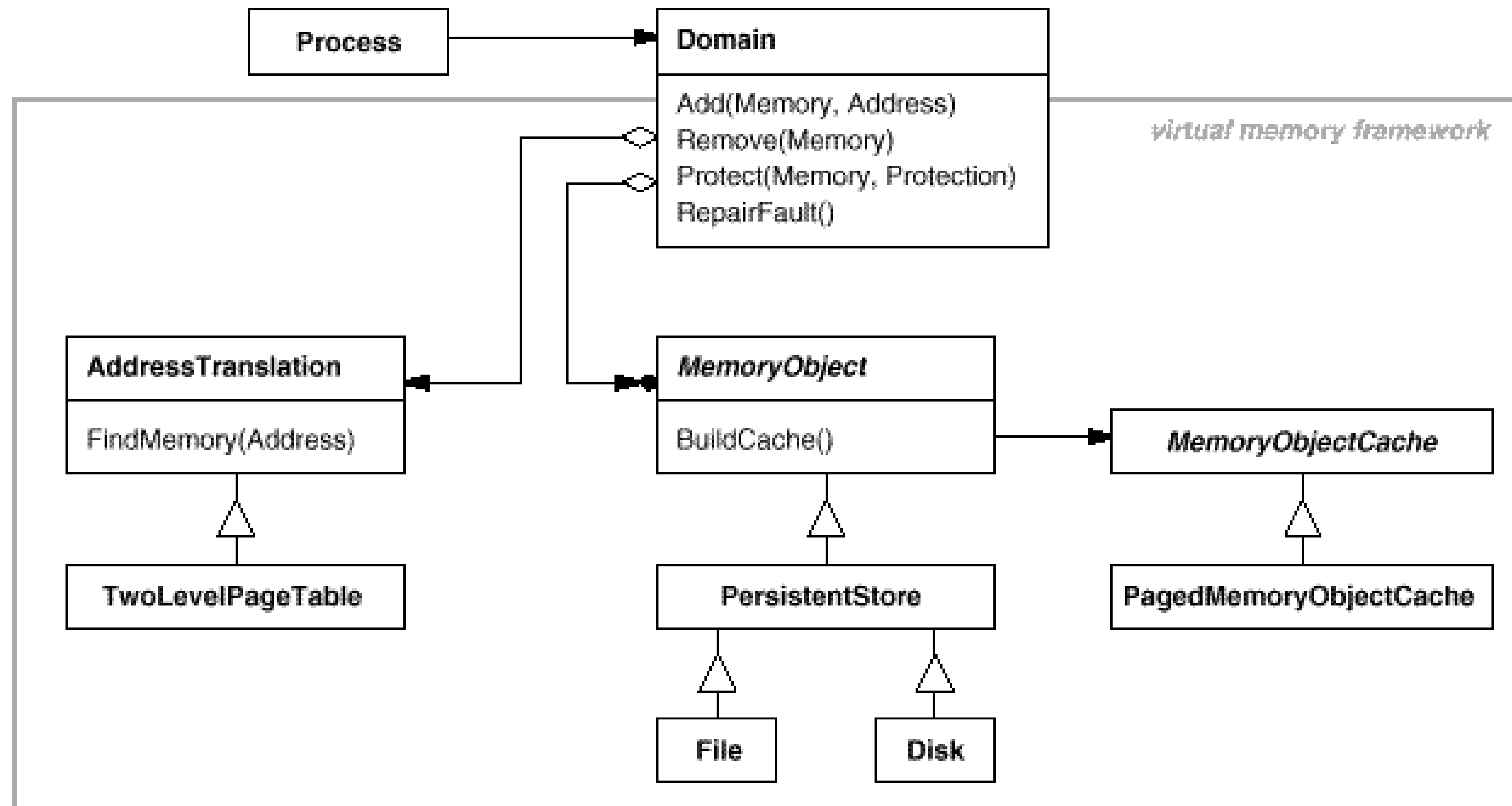
Estrutura



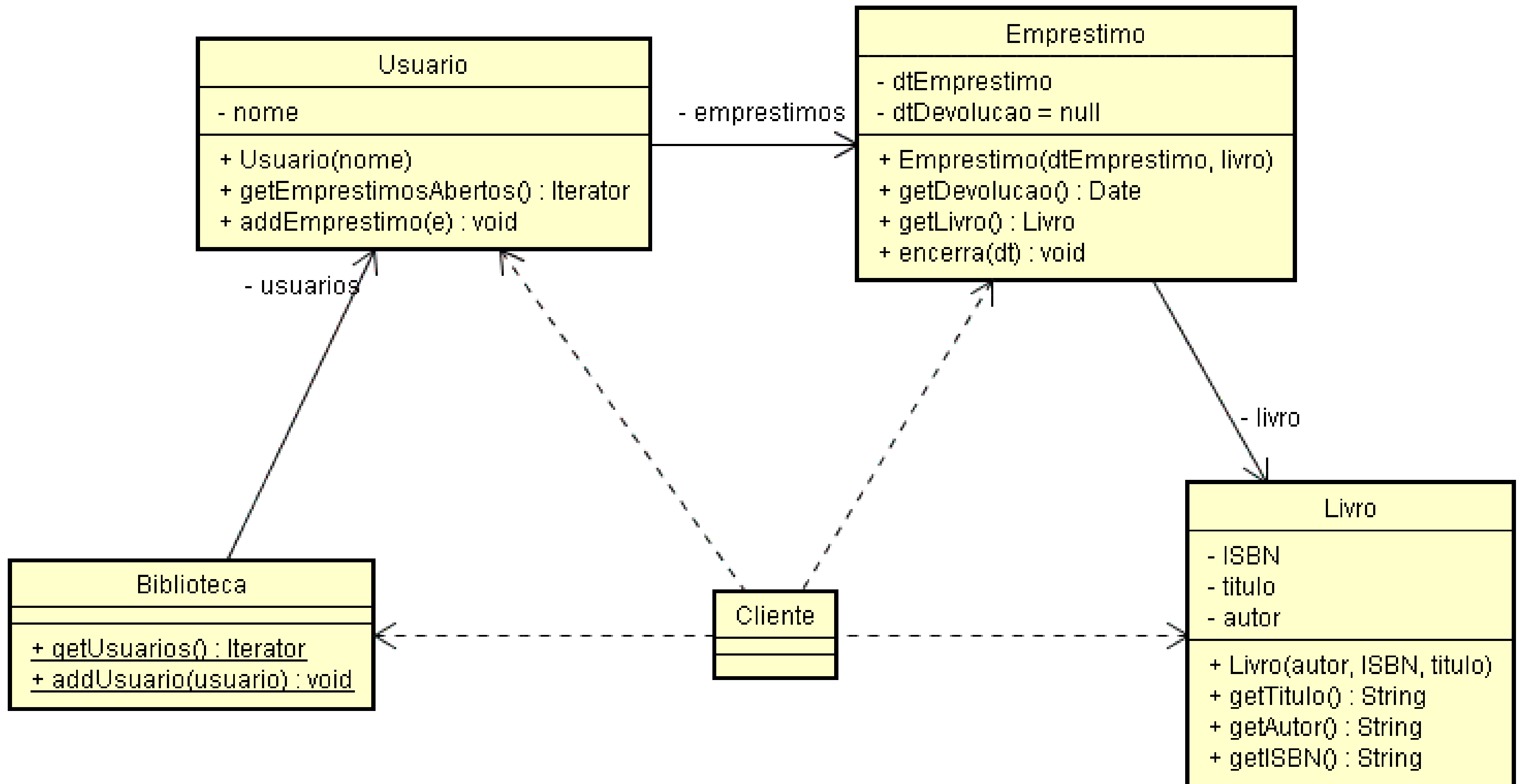
Exemplo – compilador



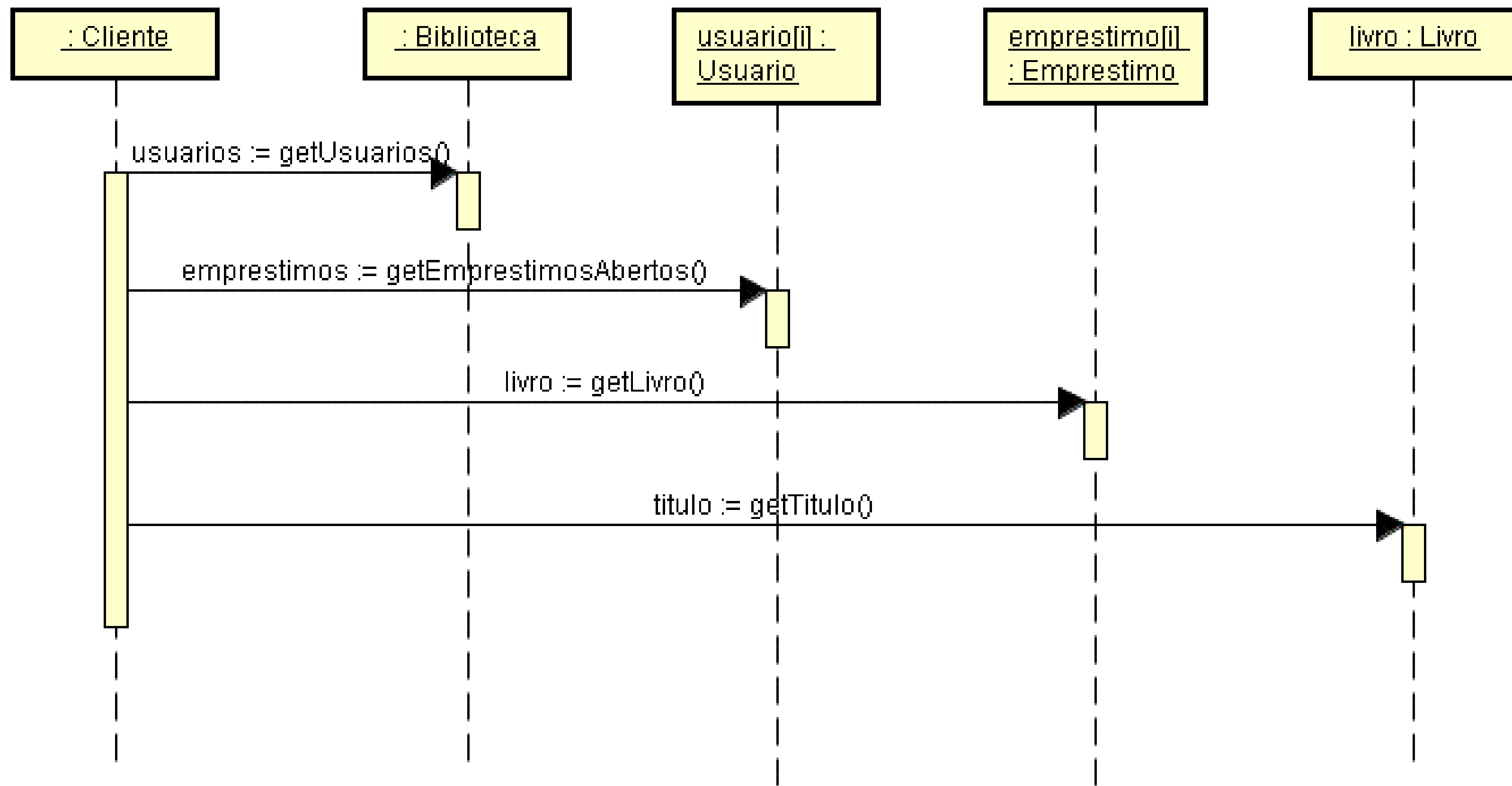
Exemplo Memória



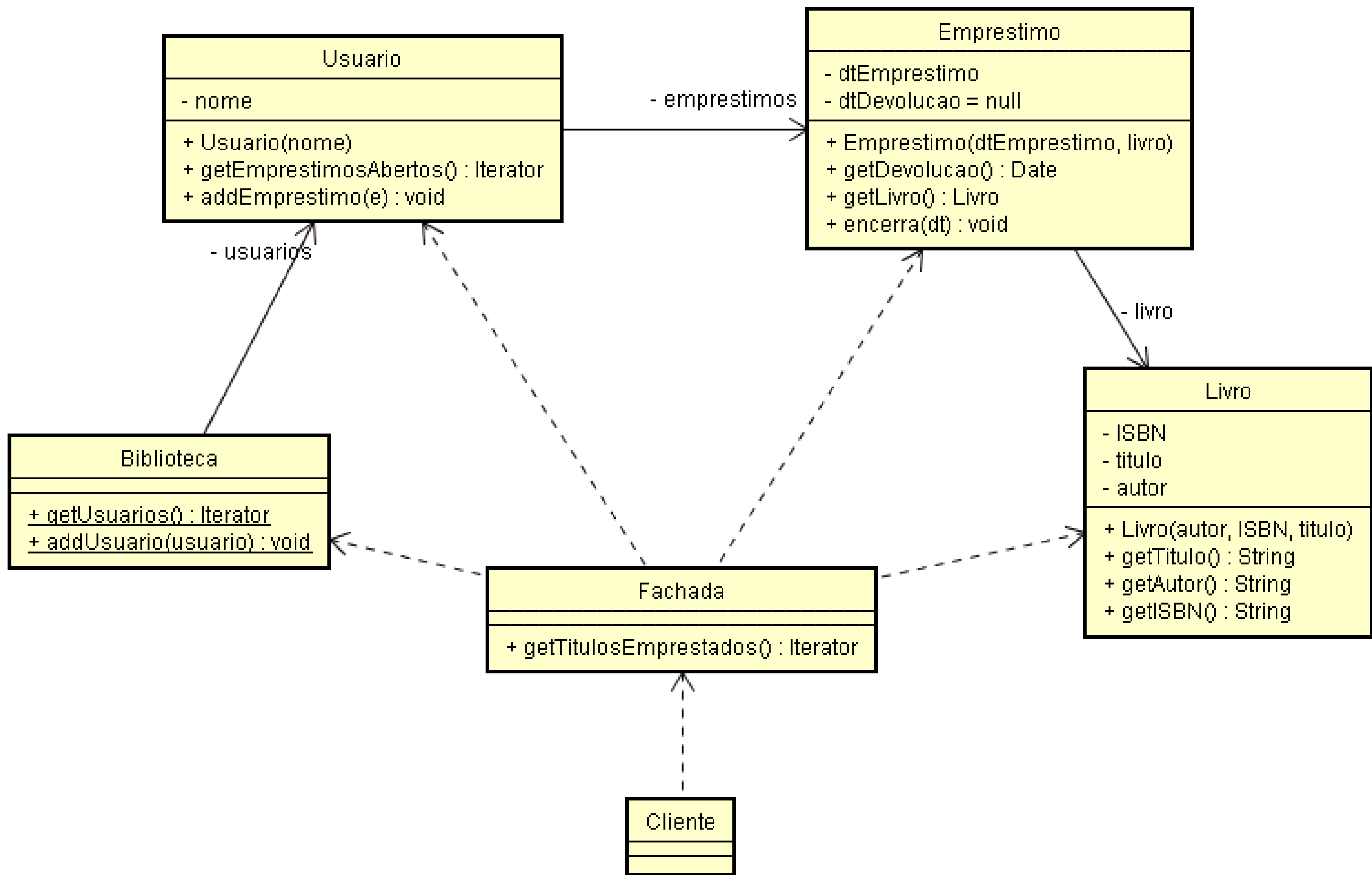
Exemplo Biblioteca: Uma implementação pobre



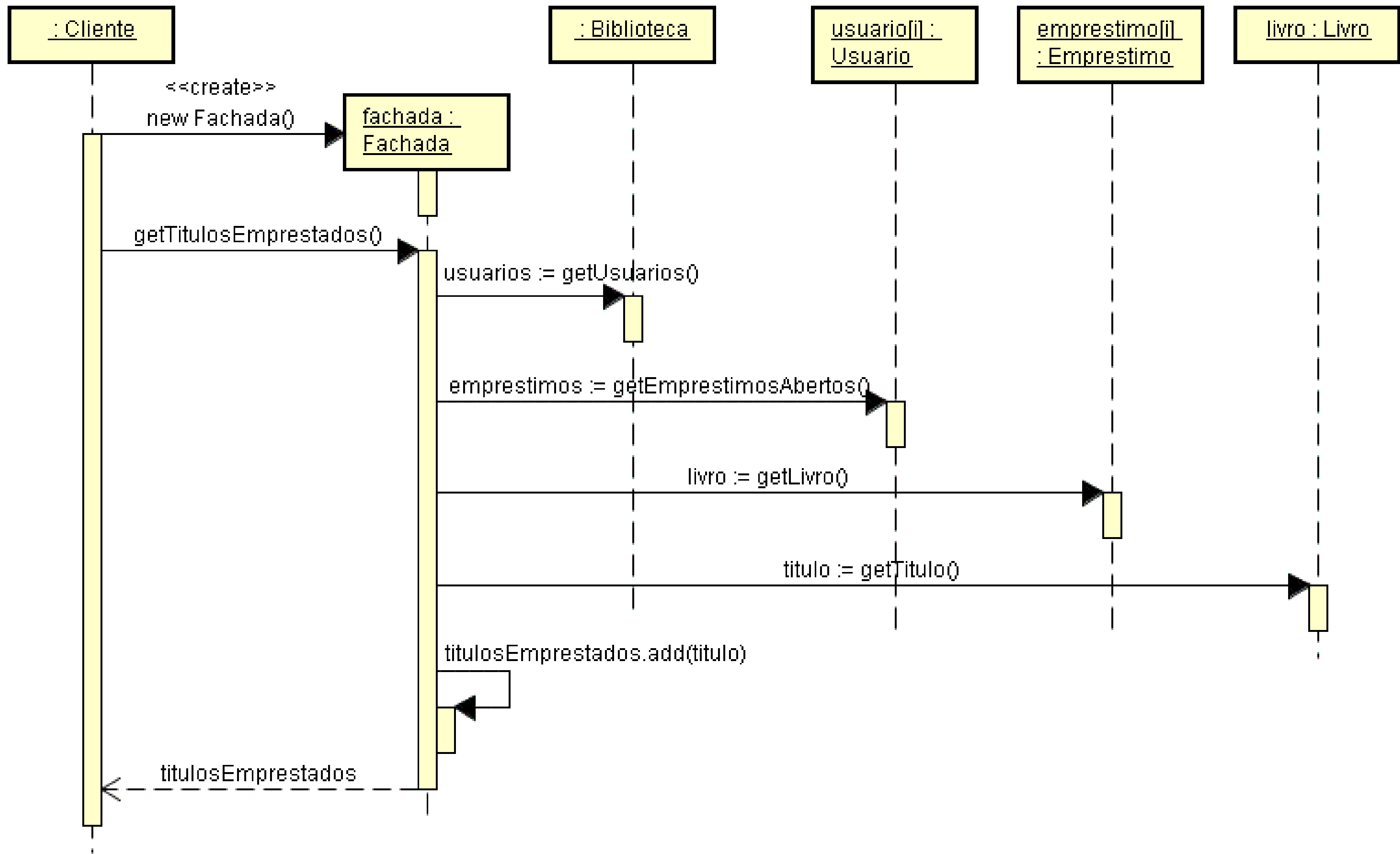
Exemplo Biblioteca: Uma implementação pobre



Exemplo Biblioteca: Uma implementação melhor



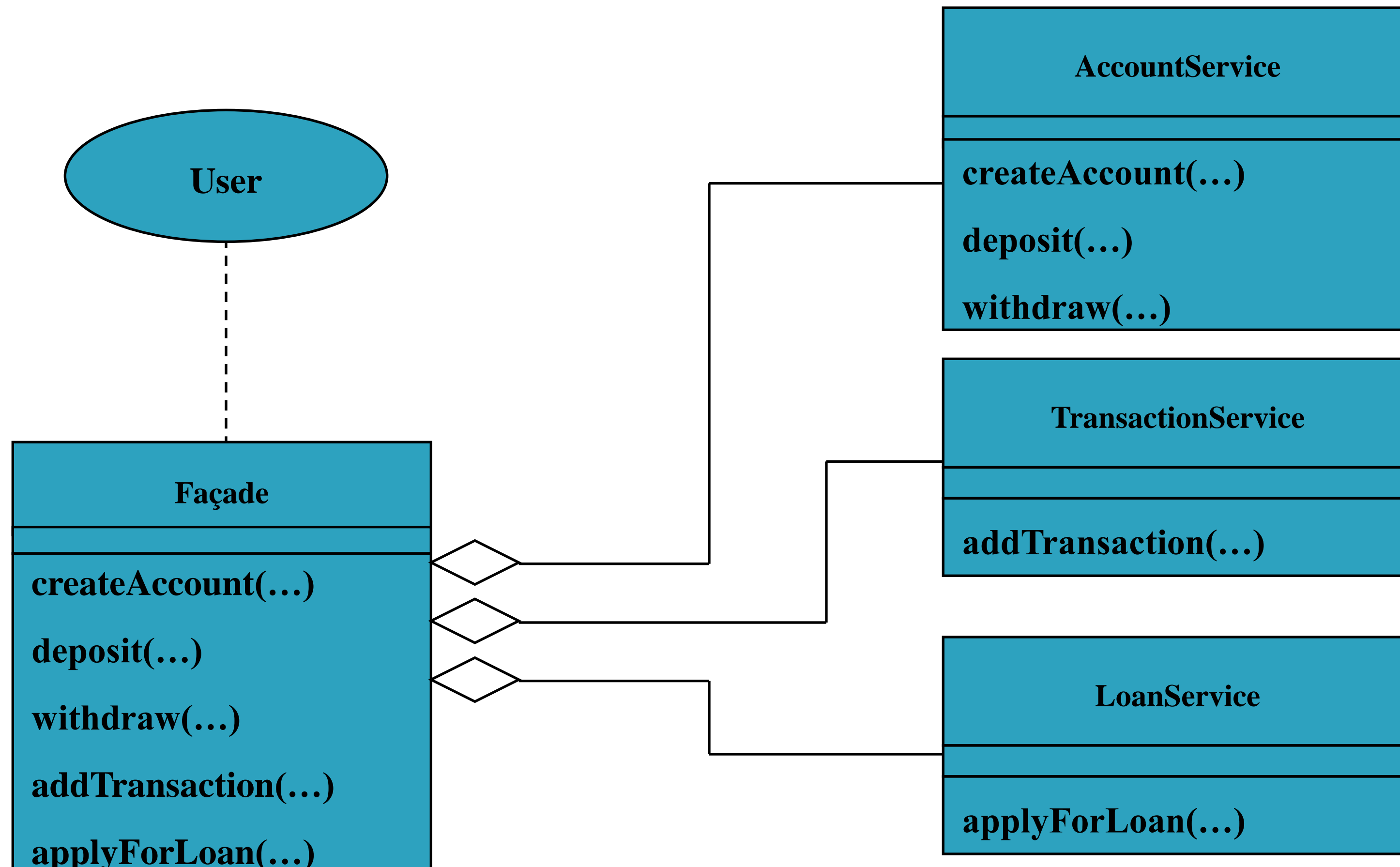
Exemplo Biblioteca: Uma implementacão melhor



Padrões relacionados

- Abstract Factory (99)
- Mediator (305)
- Singletons (144).

Exemplo Banco



Exemplo Banco (cont.)

```
public class BankFaçade {
   private AccountService accountService;
   private TransactionService transactionService;
   private LoanService loanService;

   public addAccount(Account account) {
         accountService.addAccount(account);
   }
   public deposit(int accountId, float amount) {
         accountService.deposit(accountId, amount);
   }
   public withdraw(int accountId, float amount) {
         accountService.withdraw(accountId, amount);
   }
   public addTransaction(Transaction tx) {
         transactionService.addTransaction(tx);
   }
   public applyForLoan(Customer cust, LoanDetails loan) {
         loanService.apply(cust, loan);
   }
}
```

Exemplo – Banco2

Façade em Java

```
class Aplicação {
   ...
   Facade f;
   // Obtem instancia f
   f.registrar("Zé", 123);

   f.comprar(223, 123);
   f.comprar(342, 123);

   f.fecharCompra(123);

   ...
}
```

```
public class Facade {
  BancoDeDados banco = Sistema.obterBanco();
  public void registrar(String nome, int id) {
     Cliente c = Cliente.create(nome, id);
     Carrinho c = Carrinho.create();
     c.adicionarCarrinho();
  }
  public void comprar(int prodID, int clienteID) {
     Cliente c = banco.selectCliente(cliente ID);
     Produto p = banco.selectProduto(prodID) {
     c.getCarrinho().adicionar(p);
   }
   public void fecharCompra(int clienteID) {
      Cliente c = banco.selectCliente(clienteID);
      double valor = c.getCarrinho.getTotal();
      banco.processarPagamento(c, valor);
   }
}
```

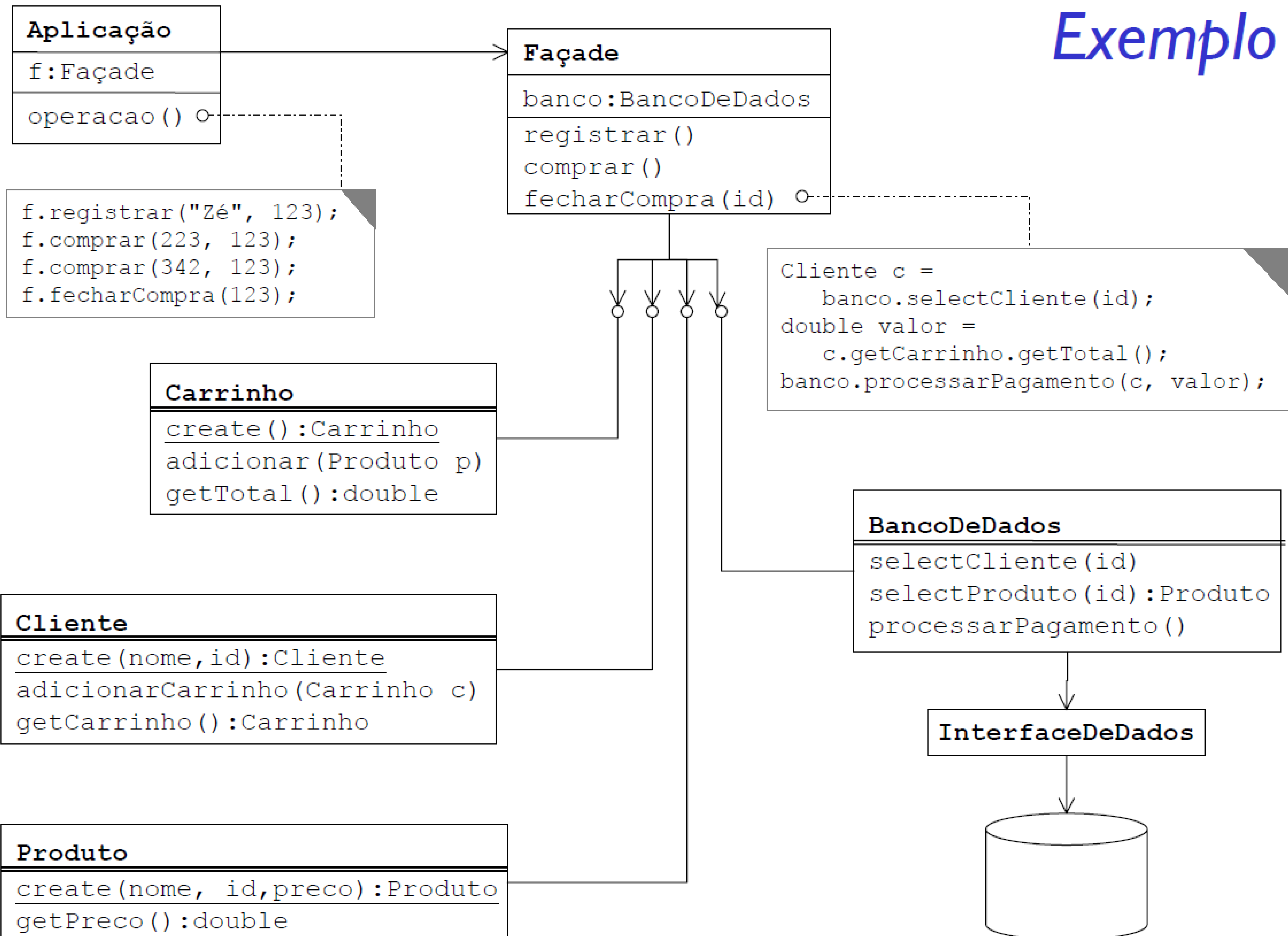
```
public class Carrinho {
  static Carrinho create() {...}
  void adicionar(Produto p) {...}
  double getTotal() {...}
}
```

```
public class Produto {
  static Produto create(String nome,
                        int id,double preco) {...}
  double getPreco() {...}
}
```

```
public class Cliente {
 static Cliente create(String nome,
                       int id) {...}
 void adicionarCarrinho(Carrinho c) {...}
 Carrinho getCarrinho() {...}
```

```
public class BancoDeDados {
 Cliente selectCliente(int id) {...}
 Produto selectProduto(int id) {...}
 void processarPagamento() {...}
}
```

Banco 2 (cont.)



Exemplo TestComputer

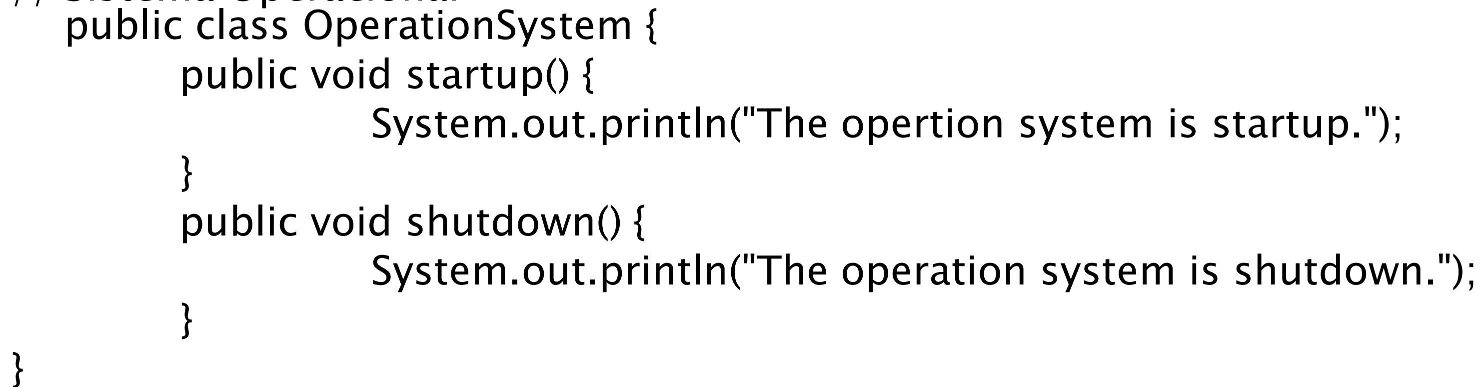
```
//FORÇA
public class Power {
   public void connect() {
          System.out.println("The power is connected.");
   }
   public void disconnect() {
          System.out.println("The power is disconnected.");
   }
}

// PLACAMÃE
public class MainBoard {
   public void on() {
          System.out.println("The mainboard is on.");
   }
   public void off() {
          System.out.println("The mainboard is off.");
   }
}
```

Exemplo TestComputer (cont)

```
//HDD HardDisk

    public class HardDisk {
            public void run() {
                    System.out.println("The harddisk is running.");
            }
            public void stop() {
                    System.out.println("The harddisk is stopped.");
            }
     }
```



```
// Sistema Operacional
    public class OperationSystem {
            public void startup() {
                    System.out.println("The opertion system is startup.");
            }
            public void shutdown() {
                    System.out.println("The operation system is shutdown.");
            }
}
```

Exemplo TestComputer (cont.)

- Façade para Test Computer

```
public class Computer {
    private Power power;
    private MainBoard board;
    private HardDisk disk;
    private OperationSystem system;
    public Computer(Power power, MainBoard board, HardDisk disk, OperationSystem system) {
            this.power = power;
            this.board = board;
            this.disk = disk;
            this.system = system;
    }
    public void startup() {
            this.power.connect();
            this.board.on();
            this.disk.run();
            this.system.startup();
    }
    public void shutdown() {
            this.system.shutdown();
            this.disk.stop();
            this.board.off();
            this.power.disconnect();
    }
}
```

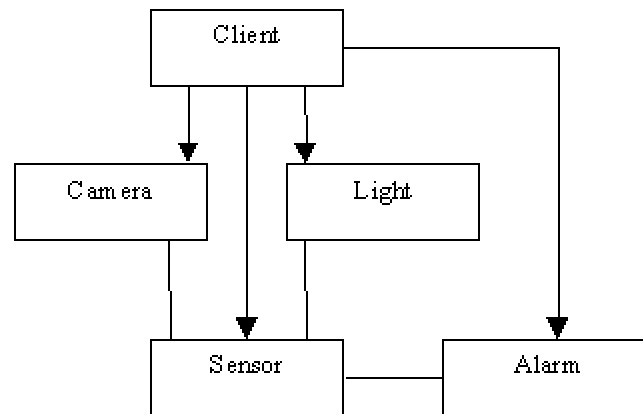
Exemplo TestComputer

- Usando o Façade para ligar e desligar o computador

```
public class TestComputer {
   public static void main(String[] args) {
         Power power = new Power();
         MainBoard board = new MainBoard();
         HardDisk disk = new HardDisk();
         OperationSystem system = new OperationSystem();
         Computer computer = new Computer(power, board, disk, system);
         computer.startup();
         computer.shutdown();
   }
}
```

Exemplo Sistema de Segurança

- **Facades e a Lei de Demeter ('Não fale com estranhos')**
- A figura mostra algumas classes que podem ser usadas em um sistema de segurança.
- Use o PP Façade para facilitar o acesso direto do cliente aos elemento.



Singleton

Ideia

- Garantir que apenas um objeto exista, independente do número de requisições que receber para criá-lo
  - Aplicações
  - Um único banco de dados
  - 1 spooler de impressão
  - Um único acesso ao arquivo de log
  - Um único objeto que representa um vídeo
  - Uma única fachada (Façade)

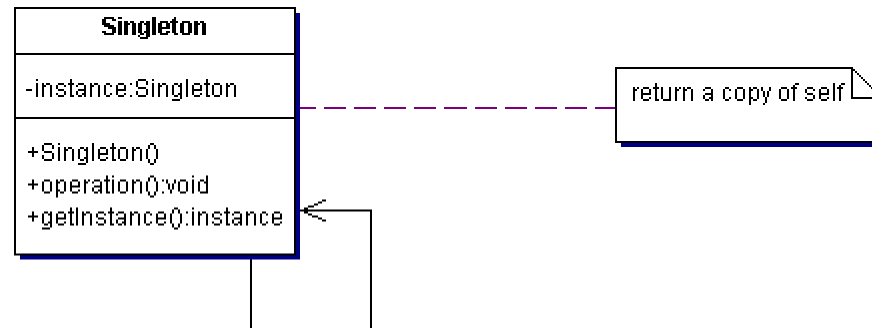
- Objetivo: garantir que uma classe só tenha uma instância

Ideia (cont)

- Uma variável global mantém um objeto acessível, mas não impede instanciar múltiplos objetos

- Solução: fazer a própria classe
  - ser responsável por criar somente 1 instância dela própria.
  - garantir que nenhuma outra instância seja criada

Estrutura



- **Participantes**
  - Define uma operação getInstance() que permite que clientes acessem sua instância única
  - É um método estático (class method)
  - Pode ser responsável pela criação de sua instância única

- **Colaborações**
  - Clientes acessam a instância apenas através da operação getInstance() do Singleton

Consequências

- Vários benefícios existem:
  - O Singleton tem controle sobre como e quando clientes acessam a instância
  - Apenas a implementação interna do Singleton precisa mudar
  - Mais flexível que métodos estáticos
    - Embora possa parecer que podemos fazer o equivalente a um Singleton com métodos estáticos, lembre que isso não permitiria o polimorfismo:

```
NomeDeClasse.xpto(); // chamada não é polimórfica
// ... versus ...
Config conf = Config.getInstance();
conf.xpto(); // essa chamada é polimórfica
```

Como assegurar que haja uma única instância?

- Uma forma é de não permitir chamadas ao construtor

- A única instância é uma instância normal de uma classe, mas a classe é escrita de forma que só 1 instância possa ser criada.
  - O construtor é protegido.
  - O cliente que tenta instanciar o Singleton diretamente obterá um erro em tempo de compilação.

Como assegurar que haja uma única instância? (cont.)

- Esconder a operação de criação de instância atrás de uma operação de classe (usando uma função estática ou método de classe)

- Essa operação tem acesso à variável que armazena a única instância, e garante que a variável é inicializada com a única instância antes de retornar o valor.

  - Garante que somente 1 instância é criada e inicializada antes do primeiro uso.

Lazy instantiation

- *With lazy instantiation, a program refrains from creating certain resources until the resource is first needed -- freeing valuable memory space*.

```
if (instance == null) {
        instance = new Singleton();
}
return instance;
```

Exemplo

```
public class Singleton {

    private static Singleton instance; // own instance

        /* protected to enable controlled subclassing */
        protected Singleton() {}

        public static Singleton getInstance() {

                // 'lazy' evaluate instance
                if (instance == null) {
                        instance = new Singleton();
                }
                return instance;
        }

        public void operation() {
                System.out.println("Singleton.operation() executing" );
        }

    }
```

Exemplo 1

```
public class SingletonTest {
    private static SingletonTest referencia;

    private SingletonTest() {
        // TODO: Código do construtor aqui.
    }

    public static SingletonTest getInstance() {
        if (referencia == null)
            referencia = new SingletonTest();
        return referencia;
    }
}

public class Clone {
    public static void main(String args[]) throws Exception {
        SingletonTest teste = SingletonTest.getInstance();
        SingletonTest clone = (SingletonObject)teste.clone();
    }
}
```

Exemplo 2

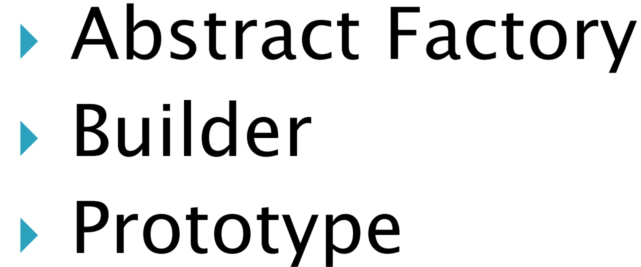
```java
public class Highlander {
  private Highlander() {}
  private static Highlander instancia = new Highlander();
  public static synchronized Highlander obterInstancia() {
       return instancia;
  }
}
```

Esta classe implementa o design pattern Singleton

```java
public class Fabrica {
  public static void main(String[] args) {
    Highlander h1, h2, h3;
    //h1 = new Highlander(); // nao compila!
    h2 = Highlander.obterInstancia();
    h3 = Highlander.obterInstancia();
    if (h2 == h3) {
       System.out.println("h2 e h3 são mesmo objeto!");
    }
  }
}
```

Esta classe cria apenas um objeto Highlander

Padrões relacionados



- Abstract Factory
- Builder
- Prototype