Análise e Desenvolvimento de Sistemas
Laboratório de Engenharia de Software

Revisão de Projeto Orientado a Objetos / Padrões de Projeto

Slides da disciplina Laboratório de Engenharia de Software
Curso de Análise e Desenvolvimento de Sistemas
Faculdade de Tecnologia de São Caetano do Sul – FATEC SCS
Prof. Marco Antonio Furlan de Souza
marco.furlandesouza@fatec.sp.gov.br

Projeto orientado a objetos

- **Para que serve a fase de projeto?**
  - **Aprofundar** em **assuntos relacionados** a **requisitos não-funcionais** e **restrições** relacionadas à **linguagens de programação**, **reutilização** de **componentes**, **sistemas operacionais**, tecnologias de **bancos de dados**, **tecnologias** de **interface homem-máquina** etc.
  - **Criar** uma **entrada apropriada** e um **ponto** de **partida** para **atividades** de **programação**, entendendo os requisitos para **subsistemas** individuais, **interfaces** e **classes**;
  - **Decompor** o **problema** em **peças gerenciáveis** para que possam ser **manipuladas** por **diferentes equipes** de desenvolvimento.

Projeto orientado a objetos

- Comparando Análise e Projeto Orientado a Objetos

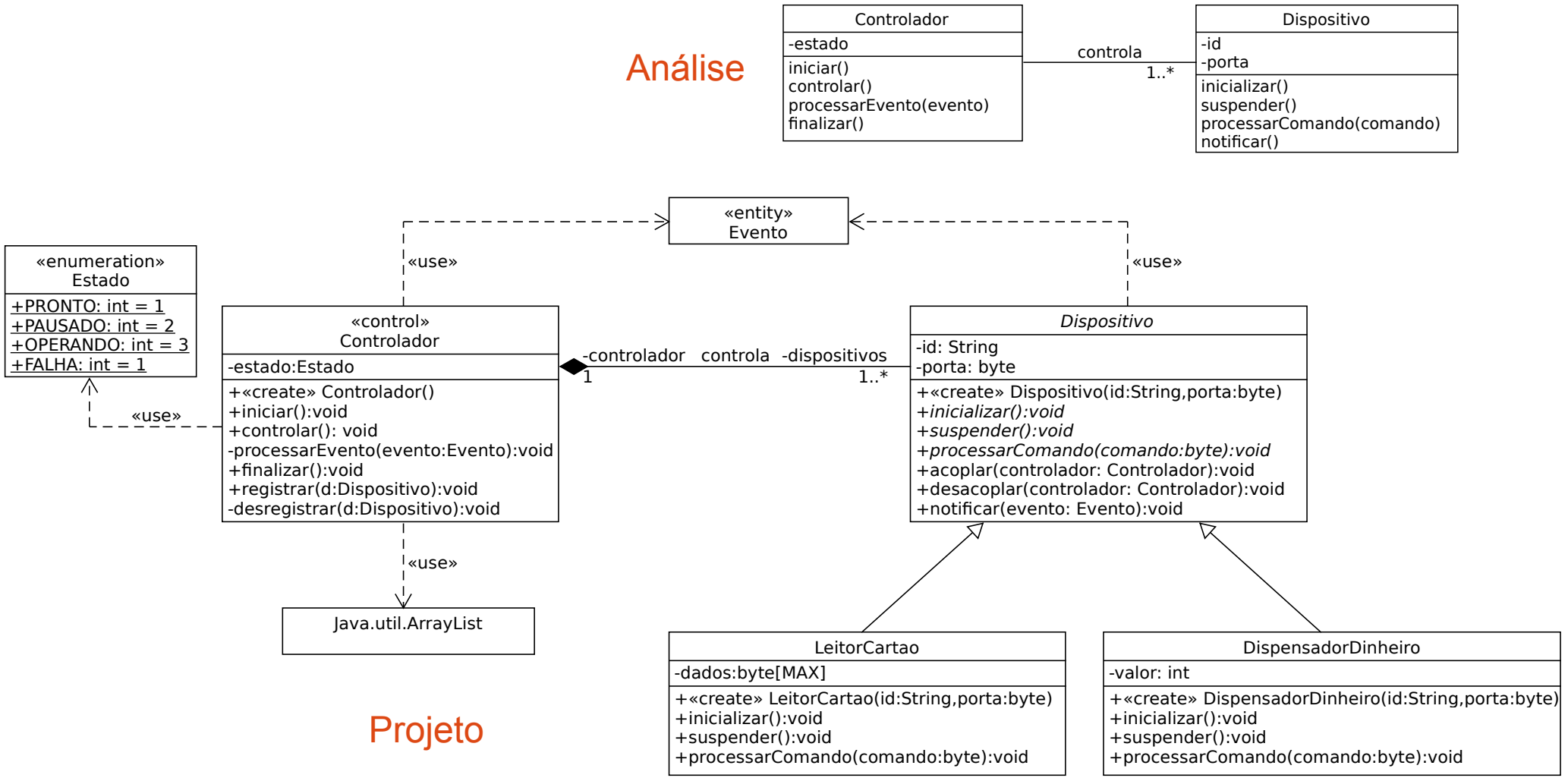
Modelos de Análise	**Modelos de Projeto**
Modelo conceitual e evita detalhes de implementação.	Modelo físico e planta para implementação.
Projeto genérico.	Específico para uma implementação.
Menos custoso para desenvolver.	Mais custoso para desenvolver.
Menos formal.	Mais formal.
Poucas camadas.	Muitas camadas.
Delineia o projeto e a arquitetura do sistema.	Implementa o projeto do sistema.
Obtido por meio de contatos constantes.	Obtido por meio de diagramação e programação visual em ambientes Integrados.
Pode não ser mantido pelo ciclo de vida completo do software.	Deve ser mantido pelo ciclo de vida completo do software.
Define a estrutura do sistema.	Tenta preservar a estrutura do sistema definido na Análise e a refina.

Projeto orientado a objetos

- **Classes de projeto**
  - Uma **classe em** um **projeto orientado** a **objetos** é uma **construção similar** àquela que se **encontrará** na **implementação** (código):
    - **Operações**, **parâmetros**, **atributos**, **tipos** seguem aqueles da **linguagem de programação** escolhida;
    - Define-se a **visibilidade final** dos **atributos** e operações de acordo com **regras da linguagem**;
    - **Decisões específicas** de **implementação** podem ser **postergadas** (com **observações** no **diagrama**);
    - **Relacionamentos** possuirão um **significado real**;
    - Pode-se **adicionar estereótipos** para **associar elementos** do **diagrama** com construções da linguagem escolhida. **Exemplo**: **«Form»**.

Projeto orientado a objetos

- **Classes de Análise e classes de Projeto**



Projeto orientado a objetos

- **Passos para executar o projeto**
  - **Particionar** o **modelo** de **análise** em **subsistemas**
    - **Separar** as **classes** em **subsistemas** de **forma** que estes possuam **interfaces** bem **definidas**, sejam **coesos** e em **número pequeno**. **Subsistemas** podem ainda ser **divididos internamente** para reduzir a complexidade;
  - **Identificar concorrência** em objetos (objetos que podem estar simultaneamente sendo utilizados por vários processos ou "threads");
  - **Alocar subsistemas** a **processadores** e **tarefas**
    - Se necessário, **utilizar mecanismos** de **concorrência** do **próprio sistema operacional** para coordenar a **concorrência** (nota: a maioria das linguagens orientadas a objetos já possui estes mecanismos).

Projeto orientado a objetos

- **Passos para executar o projeto**
  - **Desenvolver** um **projeto** para a **interface** do **usuário**;
  - **Escolher** uma **estratégia** básica para a **gestão** de **dados**: definir como será o **gerenciamento** de **dados críticos** à **aplicação** e a **infraestrutura** para **armazenar** e **recuperar** os **objetos**;
  - **Identificar** os **recursos globais** e **mecanismos** de **controle** para acessá-los;
  - **Projetar** o **mecanismo** de **controle** para o **sistema** (incluindo gerenciamento de tarefas);
  - **Considerar** como as **fronteiras** de cada **subsistema** serão **manipuladas**: determinar como as requisições entre os subsistemas serão realizadas.

Projeto da Arquitetura

- **Conceitos**
  - **Dois** tipos de arquitetura:
    - **Lógica**: organiza "**coisas**" que existem no **tempo** de **projeto** – classes e tipos;
    - **Física**: organiza "**coisas**" que existem no **tempo** de **execução** – subsistemas, componentes, tarefas e objetos.
  - **Padrões de Projeto**[1] (GAMMA et al., 1995)
    - São **estruturas recorrentes** de **projeto** que podem ser **reutilizadas** em **diversos projetos**;
    - **Padrões** são representados por **três partes principais**:
      - O **problema** que resolve – a razão de se criar o padrão;
      - A **solução** – o padrão em si;
      - **Consequências** – resultados esperados pela aplicação do padrão.

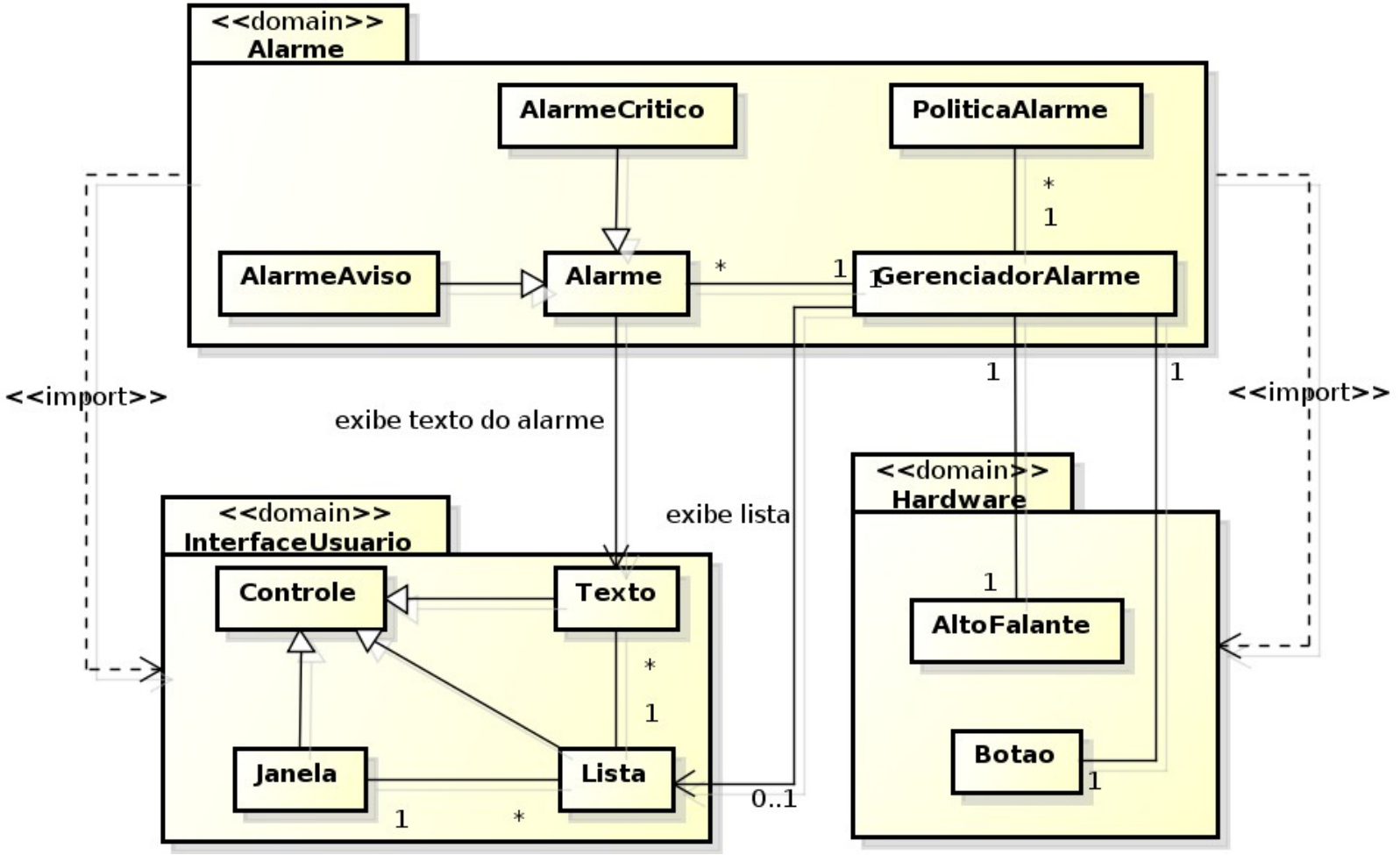
[1]Design Patterns

Projeto da Arquitetura

- **Arquitetura lógica**
  - **Separação em domínios**
    - **Área** de **conhecimento**, **independente**, com vocabulário próprio;
      - **Exemplos**: interface do usuário, hardware, gerenciamento de alarme, comunicações, sistema operacional, gerenciamento de dados, diagnósticos médicos, e assim por diante;
    - **Domínios** são **organizados** em **pacotes estereotipados UML**.

Projeto da Arquitetura

- **Arquitetura lógica**
  - **Exemplo de organização de classes com pacotes**



Projeto da Arquitetura

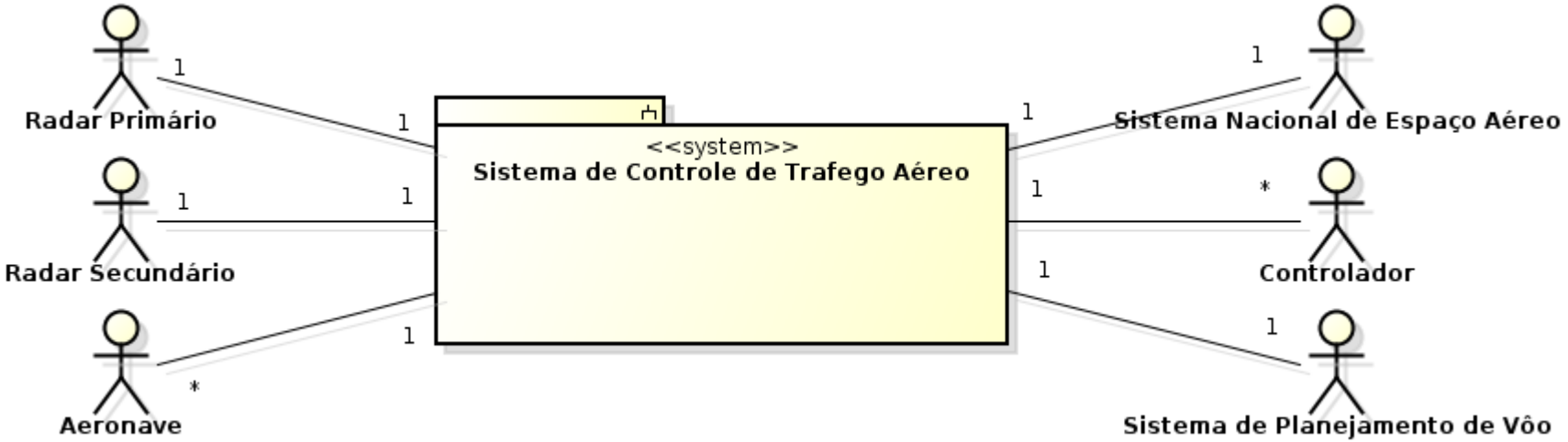
- **Arquitetura física**
  - É a **organização** em **grande escala** dos **elementos** do **sistema** em **tempo** de **execução**;
  - **Elementos típicos**:
    - **Subsistemas**;
    - **Componentes**;
    - **Objetos** tipo `<<active>>`;
    - **Formas especializadas** de **subsistemas**.

Projeto da Arquitetura

- **Visão de subsistemas**
  - Abrange **subsistemas** e **componentes**;
  - É um produto da **Engenharia** de **Sistemas**;
  - Ambos **identificam grandes partes** do **sistema** e como elas se **encaixam**;
  - **Diagrama** de **subsistemas** é um **diagrama** de **classes** que exibe os **subsistemas** e suas **relações**;
  - Em software, **subsistemas** são **grandes objetos** que contém, via **composição**, **objetos** que efetivamente **executam** as **operações** do **subsistema**;
  - Na **prática**, **subsistemas** são **implementados** por **componentes**;
  - Critério para definir **partes** de um **subsistema**: a **parte** deve **contribuir** na **realização** dos **casos** de **uso** do **subsistema**.

Projeto da Arquitetura

- **Exemplo**
  - **Representação de um sistema** (Nível 0)



Projeto da Arquitetura

- **Exemplo**
  - **Representação de um sistema** (Nível 1)

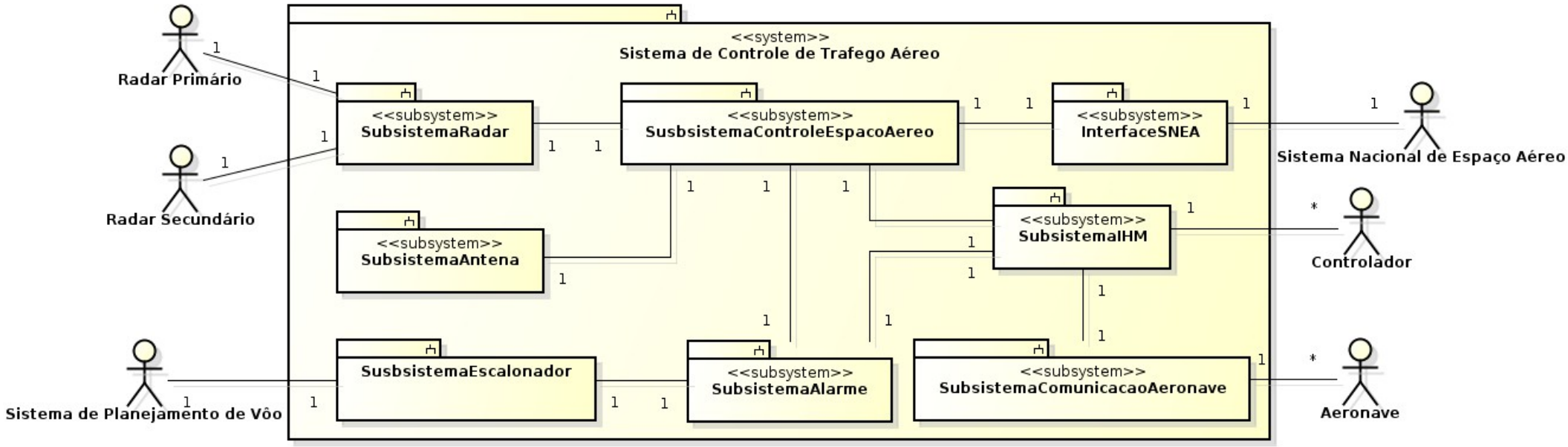


Diagrama de Componentes

- **Componentes × subsistemas**
  - **Componentes** também são **elementos** de **arquitetura**;
  - A **diferença** entre um componente e um subsistema é que o **componente** é **projetado** para ser **substituível** no sistema – **qualquer outro** componente que **honre** suas **interfaces pode** ser **utilizado** em seu lugar;
  - Uma i**nterfaces** é uma **coleção** de **operações** que possuem apenas **assinaturas** e **não implementações** (operações **abstratas**);
  - Vantagens da utilização de interfaces:
    - **Minimiza** o **acoplamento** no **sistema**;
    - **Garante** que cada **subsistema** realize **corretamente** o **comportamento especificado** por suas **interfaces**.

Diagrama de Componentes

- **Conceito de interface**
  - É um **conjunto nomeado** de **características públicas**;
  - **Interfaces** são **utilizadas** para **separar especificação** de uma **funcionalidade** (a **interface**) de sua **implementação**, por um **classificador** como **classe** ou **subsistema** (componente);
  - Uma **interface não pode ser instanciada** – ela **simplesmente declara** um **contrato** que pode ser **realizado** por **zero** ou **mais classificadores**;
  - Para que uma **interface** se **torne útil** de fato, ela **precisa ser realizada** – concretizada – por **alguma classe** que **implemente** o **corpo** de suas **operações**;
  - **Qualquer coisa** que **realize** uma **interface aceita** e **concorda** em **cumprir** o **contrato** que a interface define.

Diagrama de Componentes

- **Conceito de interface**
  - **Interfaces × realizações de interface**

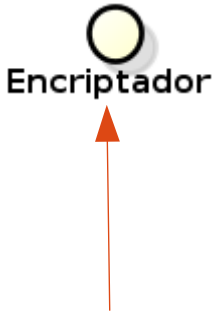
Interface especifica	Classificador que realiza
Operação	Deve ter uma operação com a mesma assinatura e semântica.
Atributo	Deve ter operações públicas, para ler e alterar os atributos da interface, embora ele não seja obrigado a ter o atributo especificado pela interface, mas ele deve se comportar como se tivesse.
Associação	Deve ter uma associação com o classificador-alvo – se uma interface especifica uma associação para outra interface, os classificadores que as implementam devem ter uma associação entre eles.
Restrição	Deve suportar a restrição.
Estereótipo	Possui o estereótipo.
Valores rotulados	Possui os valores rotulados.
Máquina de estado de protocolo.	Deve realizar este protocolo.

Diagrama de Componentes

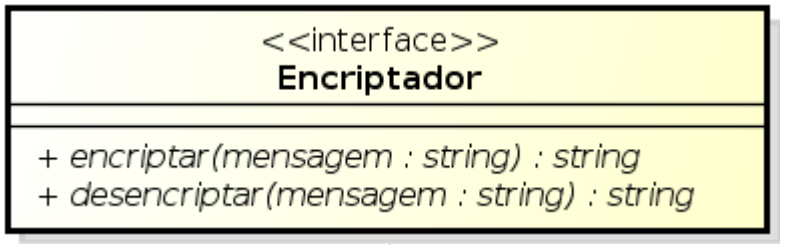
- **Conceito de interface**
  - **Interfaces × realizações de interface**
    - As **interfaces** necessitam de **especificações** da **semântica** de suas **características – texto** ou **pseudocódigo** para guiar os **implementadores**:
      - A **assinatura completa** das **operações –** nome, tipos de todos os parâmetros e tipo de retorno;
      - **Semântica** da **operação –** texto ou pseudo código;
      - **Nome** e **tipo** dos **atributos**;
      - **Qualquer esterótipo** de **atributos** ou **operações**, **restrições** e **valores rotulados**;
      - **Interfaces definem contratos**! Nunca uma implementação particular!

Diagrama de Componentes

- **Conceito de interface**
  - **Notação UML de interface**



Visão icônica



Visão completa

Diagrama de Componentes

- **Conceito de interface**
  - **Realização de interface**
    - **Interfaces** precisam ser **realizadas** por classes **concretas**;
    - Em **UML**, utiliza-se o **relacionamento de realização**:

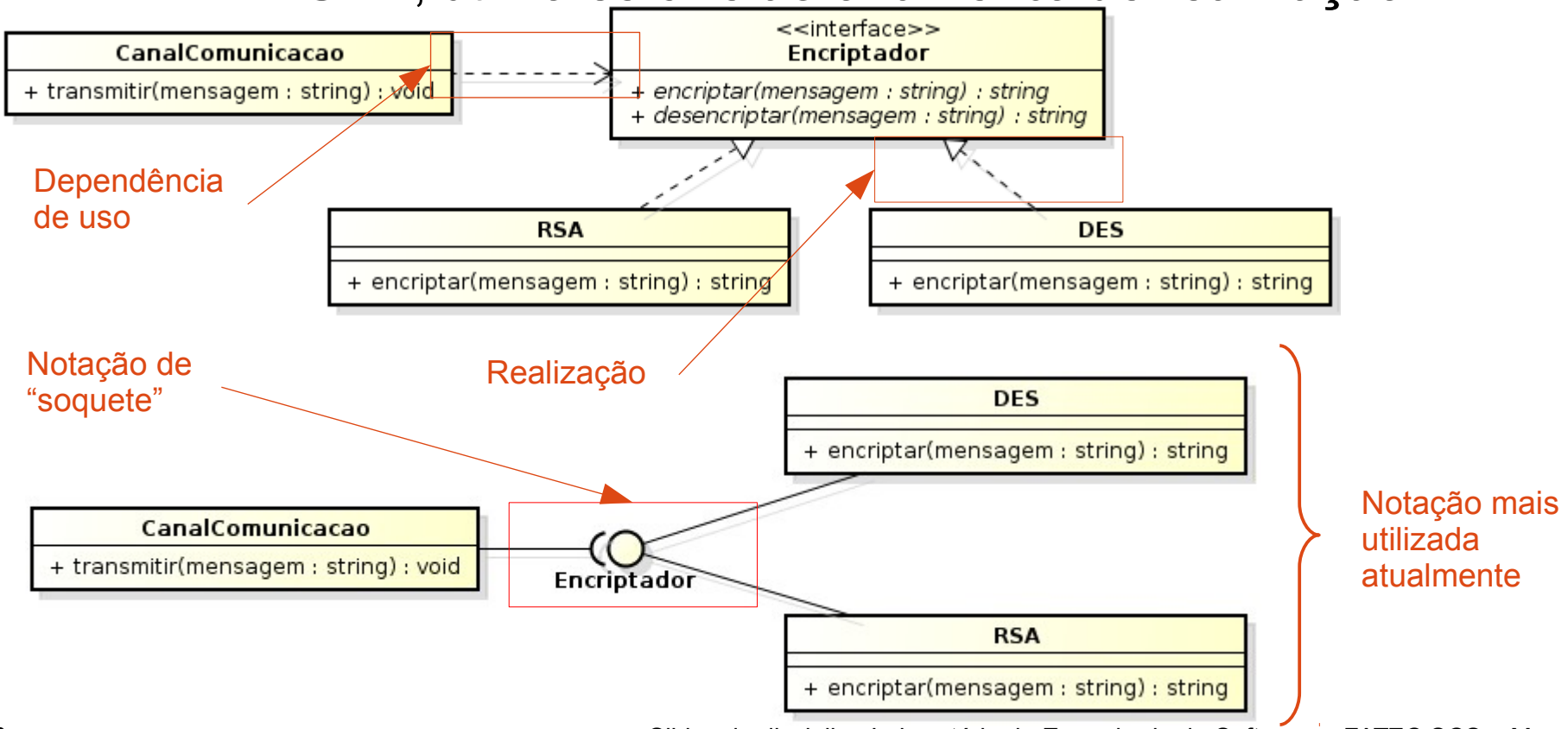


Diagrama de Componentes

- **Conceito de interface**
  - **Como as interfaces existem na prática**

```
package observer;
public interface Subject {
	// methods to register and unregister
	//observers
	public void attach(Observer obj)
		throws Exception;
	public void detach(Observer obj)
		throws Exception;
	// method to notify observers of change
	public void notifyObservers();
}
```

Interface

Em Java

Realização

```
package observer;
import java.util.ArrayList;
import java.util.List;
public class ConcreteSubject implements Subject {
	private List<Observer> observers;
	public ConcreteSubject() throws Exception {
		observers = new ArrayList<Observer>();
		if (observers == null)
			throw new Exception("null observer detected!");
	}
	public void attach(Observer obj) throws Exception {
		if (obj != null) {
			observers.add(obj);
		} else
			throw new Exception("null observer detected!");
	}
	public void detach(Observer obj) throws Exception {
		if (obj != null) {
			observers.remove(obj);
		} else
			throw new Exception("null observer detected!");
	}
	public void notifyObservers() {
		for (Observer o : observers)
			o.update(this);
	}
	public String getData() {
		return "Data from subject!";
	}
}
```

Diagrama de Componentes

- **Conceito de interface**
  - **Como as interfaces existem na prática**
    - **C++ utiliza** o **conceito** de **classe abstrata**:

```
class Subject {
    // methods to register and unregister observers
    virtual void attach(Observer *obj) = 0;
    virtual void detach(Observer *obj) = 0;
    // method to notify observers of change
    virtual void notifyObservers() = 0;
};
```

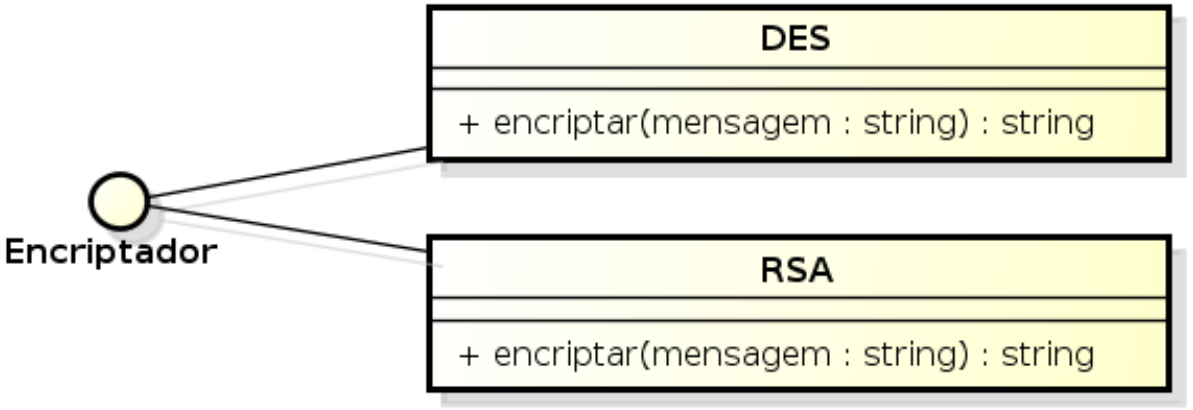
Interface

```
class ConcreteSubject: public Subject {
private:
    list<Observer*> observers;
public:
    ConcreteSubject() {
        ;
    }
    void attach(Observer *obj) {
        observers.push_back(obj);
    }
    void detach(Observer *obj) {
        observers.remove(obj);
    }
    void notifyObservers() {
        list<Observer*>::iterator i;
        for (i = observers.begin(); i != observers.end(); i++)
            (*i)->update(*this);
    }
    string getData() {
        return "Data from subject!";
    }
};
```

Implementação

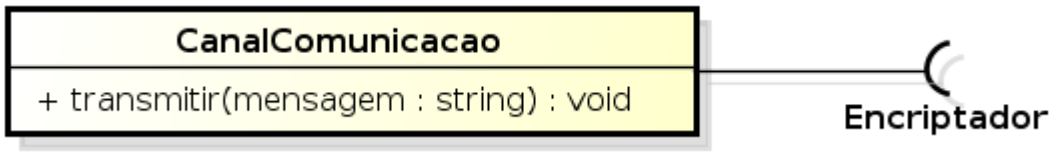
Diagrama de Componentes

- **Interfaces fornecidas e requeridas**
  - **Interface fornecida**
    - É aquela que é **realizada** pelo **classificador**.



Tanto DES quanto RSA fornecem (realizam) a interface `Encriptador`

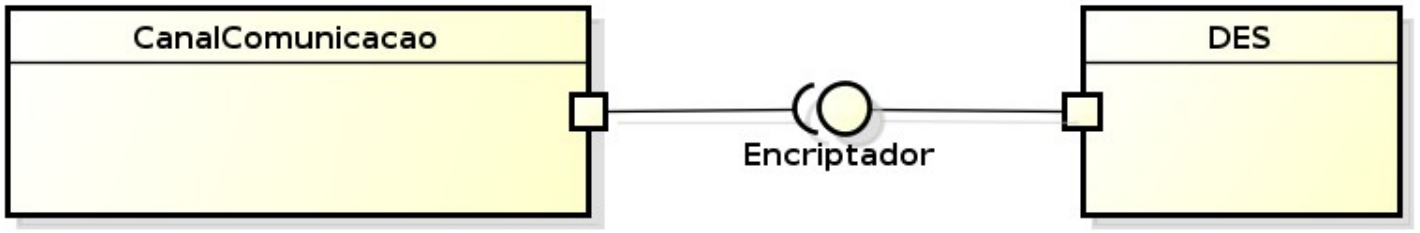
  - **Interface requerida**
    - É aquela que é o **classificador precisa** que seja **realizada**.



`CanalComunicação` requer `Encriptador` para cumprir suas responsabilidades

Diagrama de Componentes

- **Conceito de porta**
  - Uma **porta agrupa** um **conjunto semanticamente coeso** de **interfaces fornecidas** e **requeridas**:



  - O **tipo associado** a uma **porta** pode ser **explicitamente apresentado**, se a porta se referir a um **único tipo**:

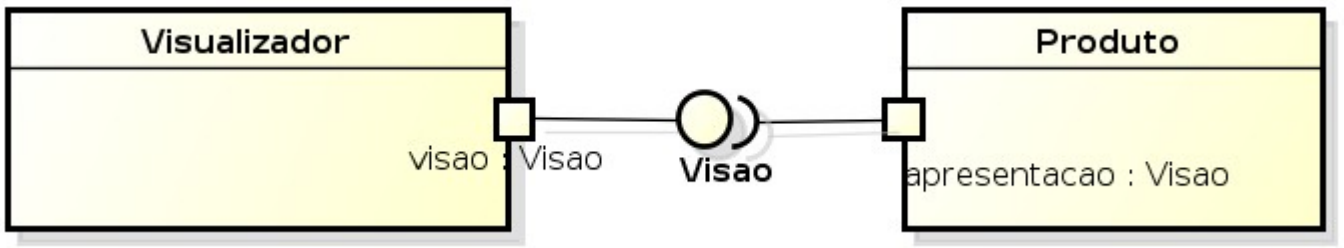


Diagrama de Componentes

- **Conceito de componente**
  - O **desenvolvimento baseado** em **componentes** baseia-se na **construção** do **software** a partir de **partes** “**plugáveis**”;
  - Essa **flexibilidade** é **proporcionada** pelo **utilização** de **interfaces**;
  - **Componentes** podem ter **atributos** e **operações** e podem **participar** de **relações** de **associação** e **generalização**;
  - Um **componente** pode ser **manifestado** por um ou mais **artefatos**.

Diagrama de Componentes

- **Conceito de componente**
  - **Sintaxe UML**

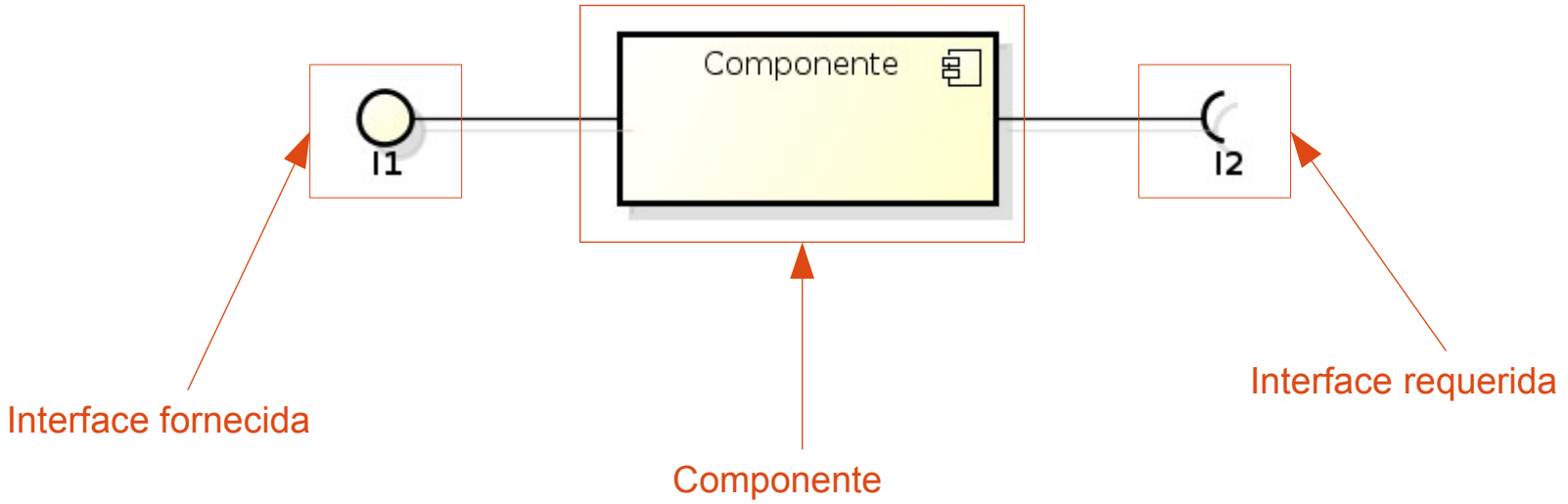


Diagrama de Componentes

- **Conceito de componente**
  - **Componentes podem ter** uma **estrutura interna – neste caso**, **delegam-se** as **responsabilidades definidas** por suas **interfaces** a **uma** ou **mais partes internas**:

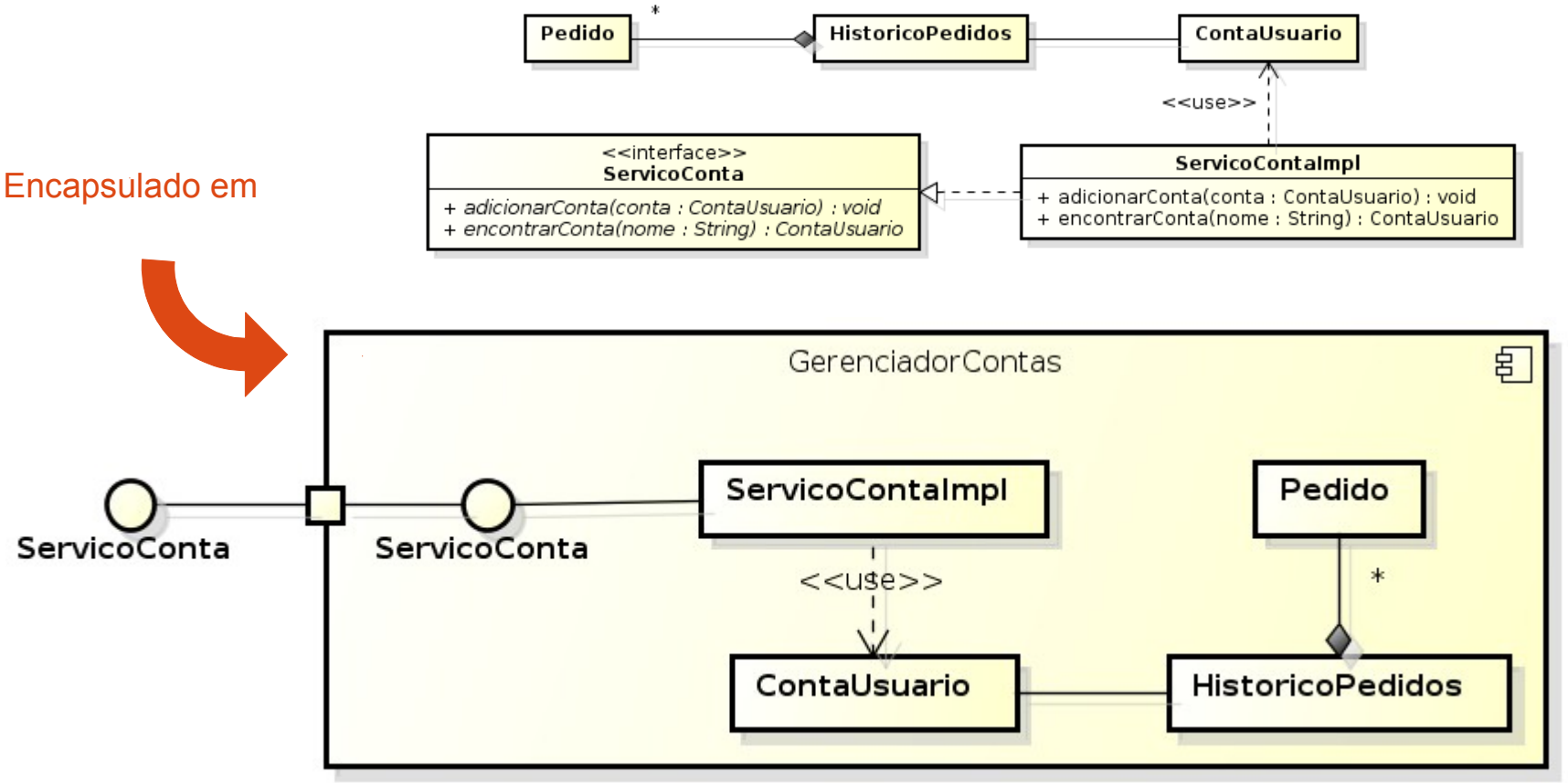
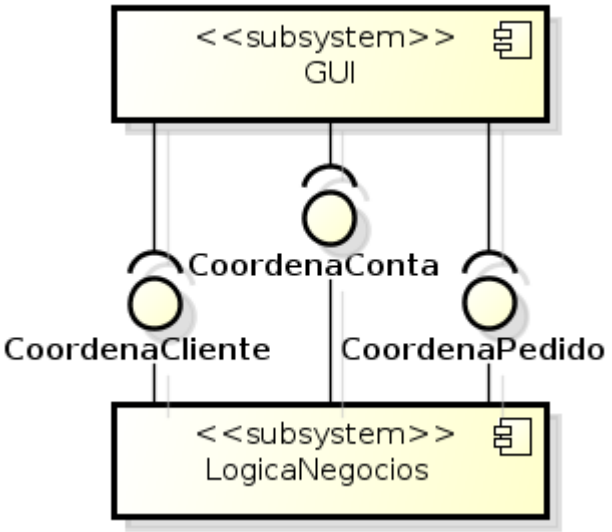


Diagrama de Componentes

- **Subsistemas como componentes**
  - **Componentes podem** ser **utilizados** para **representar subsistemas –** estereótipo `<<subsystem>>`;
  - A **diferença** é que, como **subsistemas**, o **componente não pode** ser **instanciado** em **tempo** de **execução** do **programa – mas** seu **conteúdo pode**;
  - As **interfaces conectam subsistemas** para criar uma **arquitetura** do sistema:



Padrões de Projeto

- **Conceitos**
  - Padrões de Projeto (ou Padrões de Software) são **estruturas recorrentes** que podem ser **reutilizadas** em **diversos projetos**;
  - **Objetivo**: **identificar partes** de um **mesmo projeto** que se **repetem** em **diversas instâncias** do **mesmo** e **catalogá-las** de uma **forma sistemática**;
  - **Quando** for **necessário resolver** um **problema**, o **projetista** pode **utilizar** um **padrão** como uma **solução**, **sem** ter que **redescobri-la**;
  - **Cada padrão resolve** um **tipo** de **problema** ou **trata somente** com um **tema** de **projeto**.

Padrões de Projeto

- **Classificação dos Padrões de Projeto**
  - **Padrões de Arquitetura**, representando **soluções** de **alto nível**, **reutilizáveis**, para **arquiteturas** de **software**;
  - **Padrões de Projeto**, representando **soluções reutilizáveis** de **projetos** de **software**;
  - **Padrões de Idioma**, representando **soluções padrões** de **implementação**, **dependentes** de uma **linguagem** de **programação específica**.

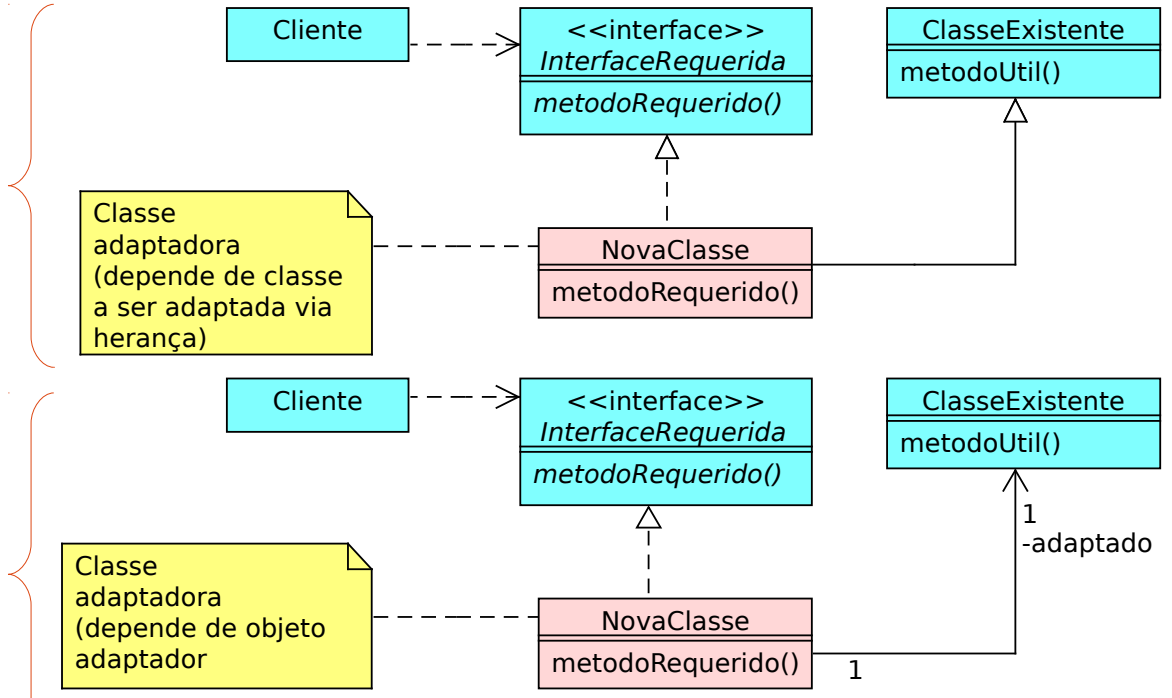
Padrões de Projeto

- **Elementos de um padrão**
  - **Nome**: **descreve**, de forma **sucinta**, o problema **relacionado** ao projeto em questão, suas **soluções** e **consequências** e permite levar-nos a um **alto grau** de **abstração**;
  - **Problema**: **descreve quando aplicar** o **padrão** – nele se **explica** o **problema** e seu **contexto**, descrevendo **requisitos específicos** do projeto;
  - **Solução**: **descreve** os **elementos** que **compõem** o **projeto**, suas **relações**, **responsabilidades** e **colaborações** em **termos gerais**. **Não descreve** uma **implementação concreta** ou **particular**, pois o padrão é uma solução geral;
  - **Consequências**: descrevem os **resultados** de se **aplicar** o **padrão**. Avaliam alternativas de projeto.

Padrões de Projeto

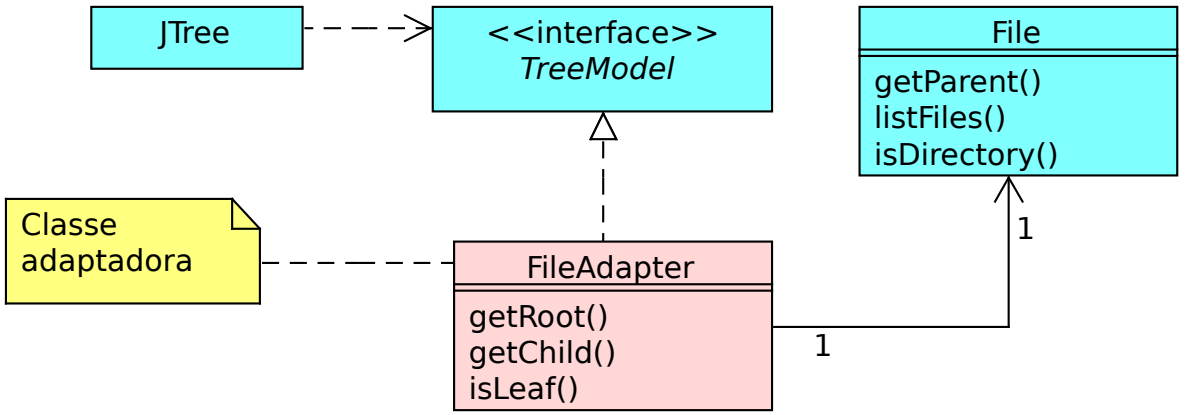
- **Exemplos**
  - **Padrão ADAPTER**
    - **Adapta** uma **interface** de classe para casar com uma **interface esperada** pelos seus clientes;
    - **Cliente** é uma **classe** que **necessita executar** um código;

Formas típicas



Padrões de Projeto

- **Exemplos**
  - **Padrão ADAPTER**
    - Exemplo de aplicação:



Padrões de Projeto

- **Exemplos**
  - **Padrão ADAPTER**
    - Exemplo de aplicação

```
package adapter;

import java.io.File;
import javax.swing.event.TreeModelListener;
import javax.swing.tree.TreeModel;
import javax.swing.tree.TreePath;

public class FileAdapter implements TreeModel {

    private File root;

    public FileAdapter(File file) {
        this.root = file;
    }

    @Override
    public Object getRoot() {
        return this.root;
    }

    @Override
    public Object getChild(Object parent, int index) {
        File[] files = ((File) parent).listFiles();
        return files[index];
    }

    @Override
    public int getChildCount(Object parent) {
        File[] files = ((File) parent).listFiles();
        if (files == null)
            return 0;
        else
            return files.length;
        }

    @Override
    public boolean isLeaf(Object node) {
        return !((File) node).isDirectory();
    }

    @Override
    public void valueForPathChanged(TreePath path, Object
        newValue) {}

    @Override
    public int getIndexOfChild(Object parent, Object child)
    {
        return 0;
    }

    @Override
    public void addTreeModelListener(TreeModelListener l)
    {}

    @Override
    public void removeTreeModelListener(TreeModelListener l)
    {}

}
```

Padrões de Projeto

- **Exemplos**
  - **Padrão ADAPTER**
    - Exemplo de aplicação

```
package adapter;

import java.awt.BorderLayout;
import java.awt.Container;
import java.io.File;

import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTree;

public class FileTreeFrame extends JFrame {
    public FileTreeFrame() {
        JTree tree;
        FileAdapter treeNode;
        Container content = getContentPane();
        File f = new File("/");
        treeNode = new FileAdapter(f);
        tree = new JTree(treeNode);
        content.add(new JScrollPane(tree), BorderLayout.CENTER);
        setSize(600, 375);
        setVisible(true);
    }
    private static final long serialVersionUID = 1L;
}
```

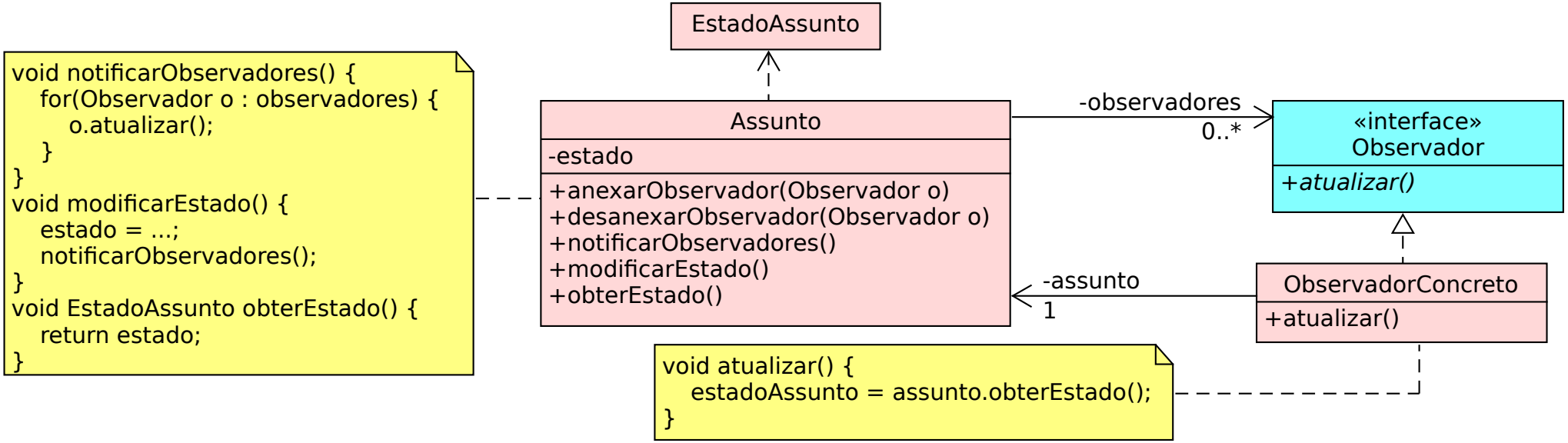
Padrões de Projeto

- **Exemplos**
  - **Padrão ADAPTER**
    - Exemplo de aplicação

```
package adapter;

public class Main {

    public static void main(String[] args) {
        new FileTreeFrame();

    }
}
```

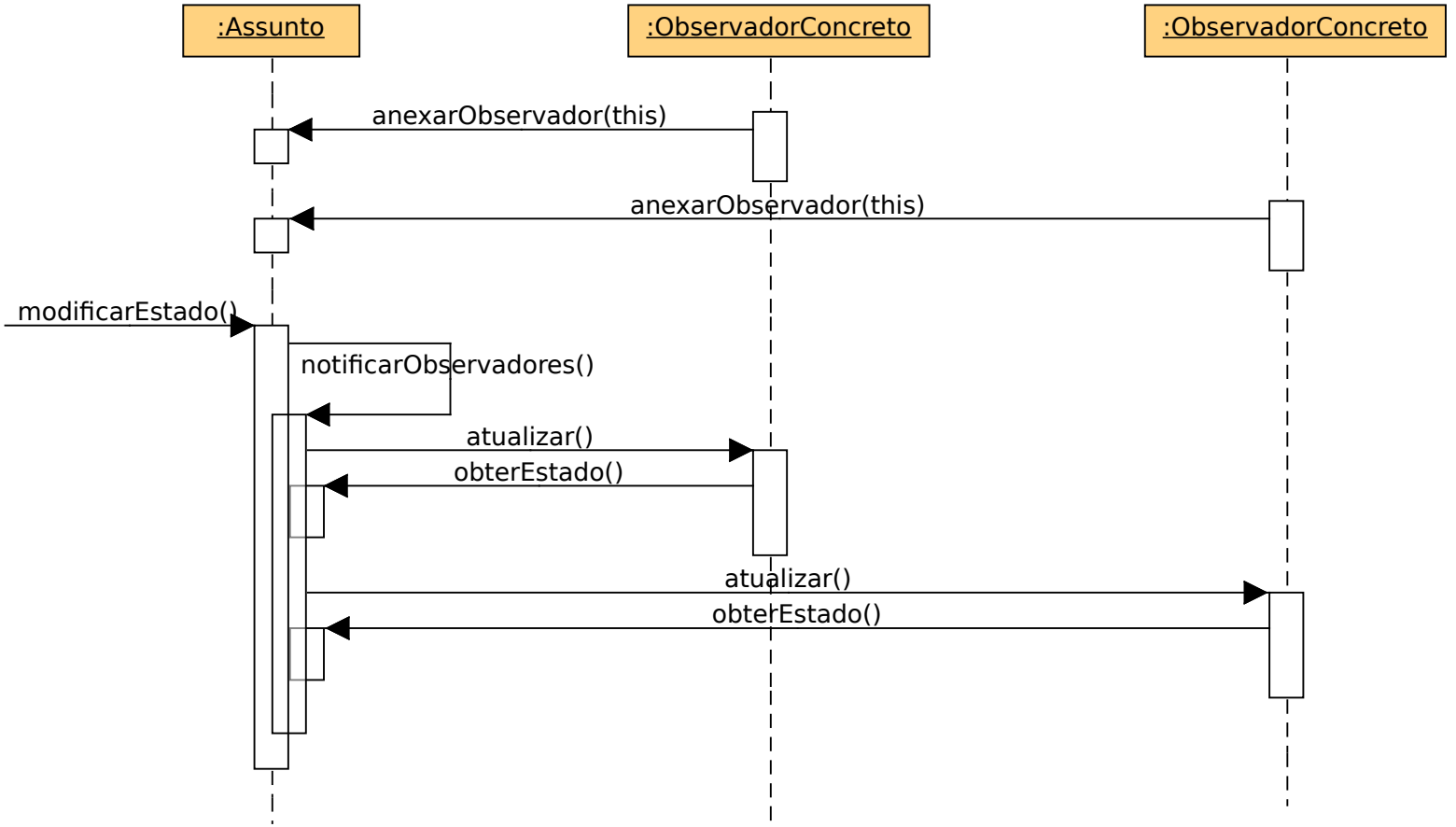
Padrões de Projeto

- **Exemplos**
  - **Padrão OBSERVER**
    - Define um **relacionamento um para vários** entre um **objeto** que faz o **papel** de "**assunto**" e quaisquer **objetos** que fazem **papel** de "**observador**", de modo que quando o **assunto** é **modificado**, os **observadores** são **notificados** e podem **reagir** às **mudanças**.



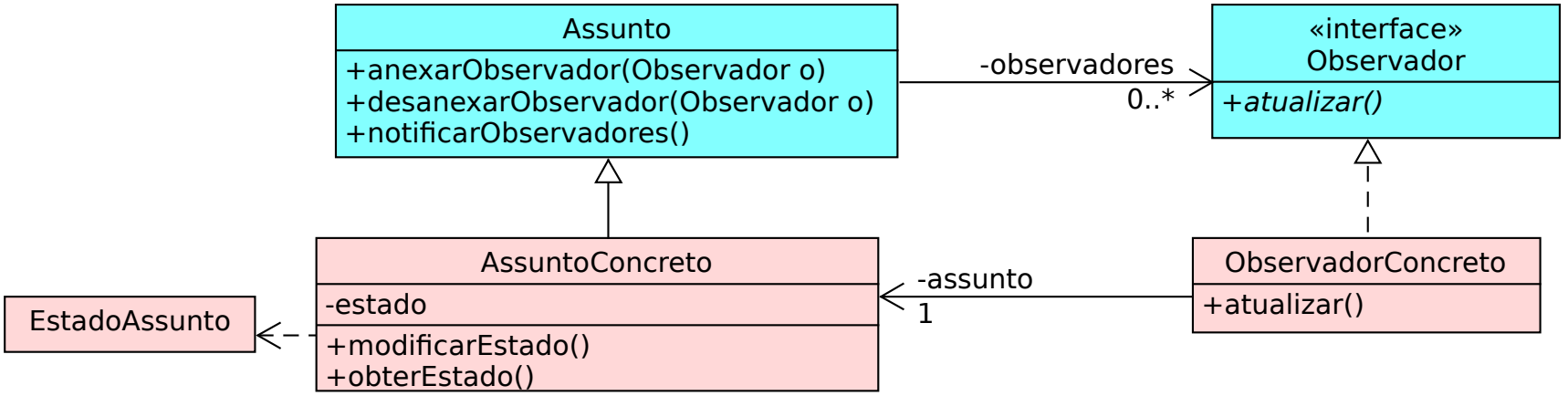
Padrões de Projeto

- **Exemplos**
  - **Padrão OBSERVER**
    - Dinâmica



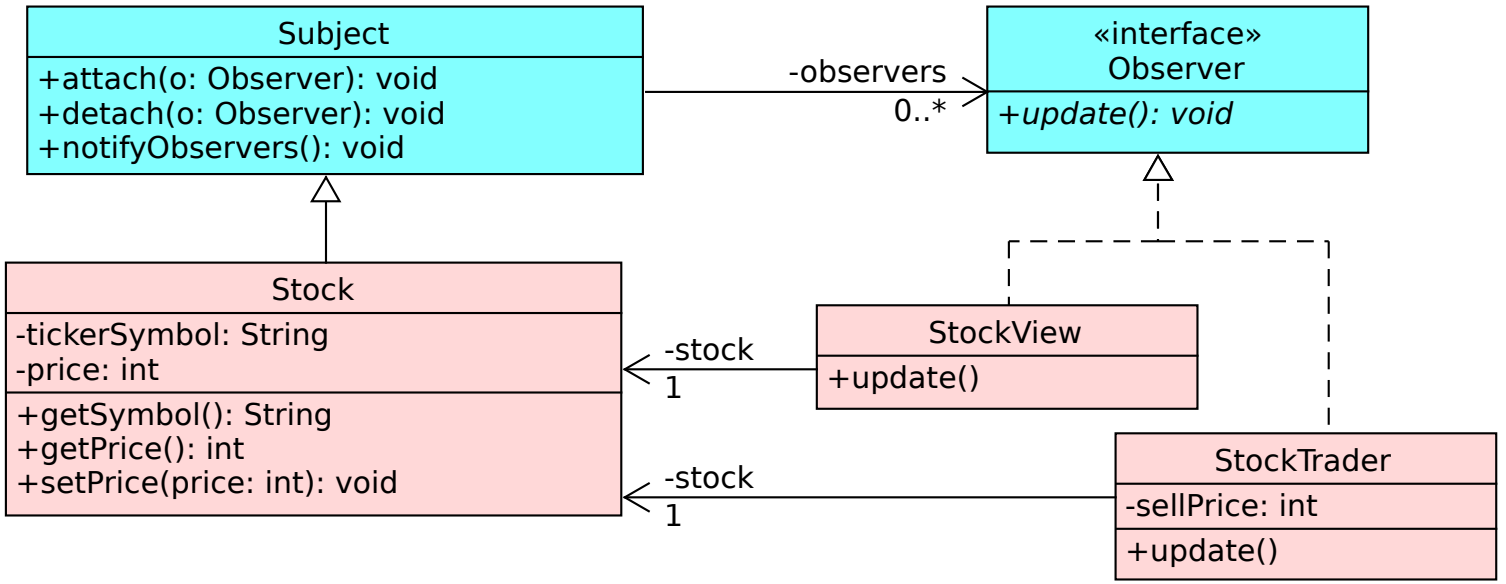
Padrões de Projeto

- **Exemplos**
  - **Padrão OBSERVER**
    - Versão alternativa



Padrões de Projeto

- **Exemplos**
  - **Padrão OBSERVER**
    - Exemplo de aplicação
      - Bolsa de valores



Padrões de Projeto

- **Exemplos**
  - **Padrão OBSERVER**
    - Exemplo de aplicação

```
package observer;
public interface Observer {
    void update();
}
```

```
package observer;
import java.util.ArrayList;

public class Subject {
    private ArrayList<Observer> observers = new ArrayList<Observer>();
    public void attach(Observer o) {
        observers.add(o);
    }
    public void detach(Observer o) {
        observers.remove(o);
    }
    public void notifyObservers() {
        for (Observer o : observers) {
            o.update();
        }
    }
}
```

Padrões de Projeto

- **Exemplos**
  - **Padrão OBSERVER**
    - Exemplo de aplicação

```
package stock;
import observer.Subject;

public class Stock extends Subject {
    private String tickerSymbol;
    private int price;
    public Stock(String tickerSymbol, int price) {
        this.tickerSymbol = tickerSymbol;
        this.price = price;
    }
    public String getSymbol() {
        return tickerSymbol;
    }
    public int getPrice() {
        return price;
    }
    public void setPrice(int price) {
        this.price = price;
        notifyObservers();
    }
}
```

- **Exemplos**
  - **Padrão OBSERVER**
    - Exemplo de aplicação

```
package stock;

import observer.Observer;

public class StockView implements Observer {
    private Stock stock;

    public StockView(Stock stock) {
        this.stock = stock;
        stock.attach(this);
    }

    @Override
    public void update() {
        System.out.println(stock.getSymbol() + " is selling for "
                           + stock.getPrice());
    }
}
```

Padrões de Projeto

- **Exemplos**
  - **Padrão OBSERVER**
    - Exemplo de aplicação

```
package stock;

import observer.Observer;

public class StockTrader implements Observer {
    private Stock stock;
    private int sellPrice;

    public StockTrader(Stock stock, int sellPrice) {
        this.stock = stock;
        this.sellPrice = sellPrice;
        stock.attach(this);
    }

    @Override
    public void update() {
        if (stock.getPrice() >= sellPrice) {
            System.out.println("Sell " + stock.getSymbol() + "!");
        }
    }
}
```

Padrões de Projeto

- **Exemplos**
  - **Padrão OBSERVER**
    - Exemplo de aplicação

```
import stock.Stock;
import stock.StockTrader;
import stock.StockView;

public class Main {
    public static void main(String[] args) {
        Stock stock = new Stock("IBM", 250);
        StockView sv = new StockView(stock);
        StockTrader st = new StockTrader(stock, 253);
        stock.setPrice(251);
        stock.setPrice(252);
        stock.setPrice(253);
    }
}
```

Referências Bibliográficas

- ARLOW, J.; NEUSTADT, I. **UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design**. Upper Saddle River, NJ: Addison-Wesley, 2005.
- BURRIS, E. **Programming in the Large with Design Patterns**. [s.l.] Pretty Print Press, 2012.
- BOOCH, G. et al. **Object-Oriented Analysis and Design with Applications**. Upper Saddle River, N.J.: Addison-Wesley, 2007.
- PRESSMAN, ROGER S. **Engenharia de Software**. McGraw-Hill Interamericana, 2002.
- PENDER, T. **UML - A Bíblia**. Campus, 2004.
- BOOCH, G. et al. **UML - Guia do Usuário**. Campus, 2000.
- SCOTT, K.; FOWLER, M. **UML Essencia**l. Bookman, 2000.
- GAMMA, Erich. **Padrões de projeto - Soluções reutilizáveis de software orientado a objetos**. Porto Alegre, RS: Bookman, 2002.