IMD0040
Linguagem de Programação 2

- Aula 22
- Design de Aplicações

Nesta aula

- Design de aplicações
  - Cap 13

- Conceitos
  - Descobrindo classes
    - Cartões CRC
  - Design de interfaces
  - Padrões de Projeto (uma introdução)

Análise e projeto

Como encontrar as classes que precisamos para resolver um problema?

- Analisamos o problema e fazemos o design (projeto) de uma solução
  - A ideia é identificar quais classes precisamos
  - E não como elas devem ser implementadas
- Usaremos duas abordagens bastante simples
  - Método verbo/substantivo para descobrir classes iniciais
  - Cartões CRC para realizar o design inicial da aplicação

O método verbo/substantivo

- Substantivos descrevem ‘coisas’
  - Fonte de classes e objetos

- Verbos descrevem ‘ações”
  - Uma fonte de interações entre objetos
  - Ações são comportamentos
    - Métodos

Exemplo de descrição de um problema

- O sistema de reservas de lugares deve armazenar reservas para múltiplas salas de cinema.
- Cada sala de cinema tem poltronas organizadas em filas.
- Os clientes podem reservar poltronas e recebem um número de fila e um número de assento.
- Eles podem reservar várias poltronas próximas (assentos contíguos).
- Cada reserva é para uma sessão particular (isto é, a exibição de um dado filme em uma determinada data e hora).
- Os filmes ocorrem em uma data e hora específicas e são programados em uma sala onde são exibidos.
- O sistema armazena o número de telefone do cliente.

Quais são as classes para implementar esse sistema?

Substantivos e Verbos

Substantivos

- Sistema de reserva de cinema
- Cinema
- Poltrona
- Fila
- Cliente
- Número da fila
- Número da Póltrona
- Sessão
- Data
- Hora
- Número de telefone

Verbos

- Armazena reservas
- Armazena número de telefone
- Tem poltronas
- Reserva Poltrona
- Recebe número da fila e da poltrona
- Solicita reserva
- É agendado (no cinema)

Substantivos e Verbos

Sistema de reservas de cinema
Armazena (reservas de poltronas)
Armazena (nº de telefone)

Sala
Tem (poltronas)

Filme

Cliente
Reserva (poltronas)
Recebe (nº da fila e do assento)
Solicita (reserva de poltronas)

Hora

Data

Reserva de poltrona

Filme
Está agendado (na sala)

Poltrona

nº da poltrona

nº do telefone

Fila

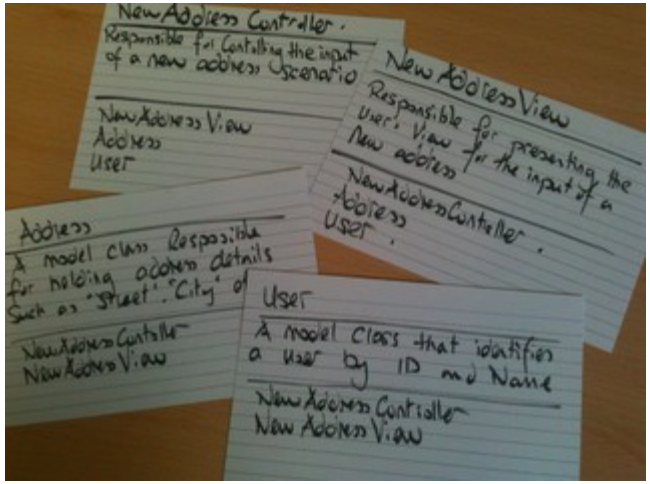
nº da fila

Utilizando cartões CRC

- Class/Responsibility/Collaboratorion
  - Descritos por Kent Beck e Ward Cunningham
- Objetivo é identificar as interações entre as classes
- Cartões de papelão (ex: cartões de índice de bibliotecas)
  - Nome da classe
  - Responsabilidades da classe
  - Colaboradores da classe
- *É Importante utilizar um cartão para cada classe*

Cartão CRC

Class Name	
Responsibilities	Collaborators



ContaBancária (entidade)	
Responsabilidades	**Colaboradores**
1.Conhecer o seu cliente. 2.Conhecer o seu número. 3.Conhecer o seu saldo. 4.Manter um histórico de transações. 5.Aceitar saques e depósitos.	Cliente HistóricoTransações

Os cartões são preenchidos iterativamente (primeiro só o nome da classe, ….)

Cenários

Um exemplo de uma atividade que o sistema tem de executar ou suportar

- Também conhecidos como casos de uso
  - Ajudam a descobrir as interações
- Podem ser realizados em grupo
  - Cada pessoa do grupo é atribuído uma classe
  - Cada pessoa fala o que a classe está fazendo atualmente
  - Cada pessoa anota as informações que forem descobertas em seu cartão

Um exemplo (parcial) de cenário

Cliente telefona para o cinema querendo reservar dois assentos para assistir a um filme hoje a noite.

- Quem é o usuário do sistema?

Um exemplo (parcial) de cenário

Cliente telefona para o cinema querendo reservar dois assentos para assistir a um filme hoje a noite.

- Quem é o usuário do sistema?
  - Funcionário do cinema

Um exemplo (parcial) de cenário

Cliente telefona para o cinema querendo reservar dois assentos para assistir a um filme hoje a noite.

- Quem é o usuário do sistema?
  - Funcionário do cinema
- Classe CinemaBookingSystem (representa o sistema)
  - Quais sessões serão apresentadas hoje a noite?

Um exemplo (parcial) de cenário

Cliente telefona para o cinema querendo reservar dois assentos para assistir a um filme hoje a noite.

- Quem é o usuário do sistema?
  - Funcionário do cinema
- Classe CinemaBookingSystem (representa o sistema)
  - Quais sessões serão apresentadas hoje a noite?
    - Responsabilidade: Poder encontrar sessão por dia ou título
    - Colaborador: Sessão

Um exemplo (parcial) de cenário

Cartão CRC

CinemaBookingSystem	
Responsabilidades:	Colaboradores:
Podem localizar filmes por título e dia	Sessão

Um exemplo (parcial) de cenário

Cliente telefona para o cinema querendo reservar dois assentos para assistir a um filme hoje a noite.

- Quem é o usuário do sistema?
  - Funcionário do cinema
- Classe CinemaBookingSystem (representa o sistema)
  - Quais sessões serão apresentadas hoje a noite?
    - Responsabilidade: Poder encontrar sessão por dia ou título
    - Colaborador: Sessão
  - Como o cinema encontra a sessão?

Um exemplo (parcial) de cenário

Cliente telefona para o cinema querendo reservar dois assentos para assistir a um filme hoje a noite.

- Quem é o usuário do sistema?
  - Funcionário do cinema
- Classe CinemaBookingSystem (representa o sistema)
  - Quais sessões serão apresentadas hoje a noite?
    - Responsabilidade: Poder encontrar sessão por dia ou título
    - Colaborador: Sessão
  - Como o cinema encontra a sessão?
    - Responsabilidade: Armazena uma coleção de sessões
    - Colaborador: Collection

Um exemplo (parcial) de cenário

Cartão CRC

CinemaBookingSystem	
Responsabilidades:	Colaboradores:
Podem localizar filmes por título e dia	Sessão
Armazena coleção de filmes	Collection

Um exemplo de cenário

- Exemplo completo está na seção 13.1.5 do livro

Cenários como análise

- Cenários servem para verificar se a descrição do problema é clara e completa
- Consome bastante tempo
- A análise resultará no projeto
  - Indicação ou omissões de erros aqui pouparão muito trabalho mais tarde
- Situações possíveis
  - Adicionar novas classes
  - Classes não utilizadas

Projeto (design) de classe

- A análise do cenário ajuda a esclarecer a estrutura da aplicação
  - Cada cartão mapeia para uma classe
  - As colaborações revelam cooperação de classe/interação de objeto
- As responsabilidade revelam
  - Métodos públicos
  - Campos (“armazena a coleção de ...”)
- ***Porém, você ainda não está pronto para ir para o código!***

Projetando interfaces de classe

- Interface de uma classe
  - “conjunto de métodos públicos”
    - Assinatura completa do método
- Como fazer?
  - Reproduza os cenários em termos de chamadas de método, valores de parâmetros e de retorno
  - Anote as assinaturas resultantes
  - Crie estruturas de classe com stubs de métodos públicos
    - Stubs: métodos com assinatura correta e corpo vazio

Documentação

- Escreva comentários de classe
- Escreva comentários de método
- Descreva o propósito geral de cada classe
- Documentar agora assegura que:
  - O foco está no ***que*** em vez de em ***como***
  - Que ele não é esquecido!

Cooperação

- Desenvolvimento de software é normalmente feito em equipes
- Documentação é essencial para o trabalho em equipe
- Projeto OO claro, com componentes fracamente acoplados, auxilia nessa cooperação
  - Cada membro da equipe pode se concentrar em uma única classe de maneira independente

Prototipagem

"Não tente construir o sistema todo de uma vez"

Trabalhe com protótipos

- Permite a investigação inicial de um sistema
  - Identificação de problemas mais cedo
- Componentes incompletos podem ser simulados
  - Retornar um resultado fixo
  - Evita comportamento aleatório

Crescimento de software

- Existem diversos modelos de como o software deve ser construído
- Ex: modelo em cascata (waterfall model)
  - Análise
  - Projeto (design)
  - Implementação
  - Teste de unidade
  - Teste de integração
  - Entrega
- Nenhuma provisão para iteração

Desenvolvimento iterativo

- Usa prototipagem mais cedo
- Interação frequente com o cliente
- Iteração por:
  - Análise
  - Projeto (design)
  - Prototipação
  - Feedback do cliente

- Um “modelo de crescimento”
  - Desenvolva uma versão simplificada do software
  - Adicione funcionalidade

Crescimento de software

- O livro favorece uma abordagem iterativa

- Habilidades trabalhadas focam nisso
  - Manutenção de software
  - Leitura de código
  - Design para extensibilidade
  - Documentação

Utilizando padrões de projeto

- Definir a estrutura do sistema é mais difícil do que escrever código para uma simples classe
  - Relacionamentos interclasses podem ser bastante complexos
- Alguns relacionamentos recorrem em diferentes aplicações
  - Reuso é interessante
- Padrões de projeto ajudam a esclarecer relacionamento e promovem o reuso

Padrões de projeto

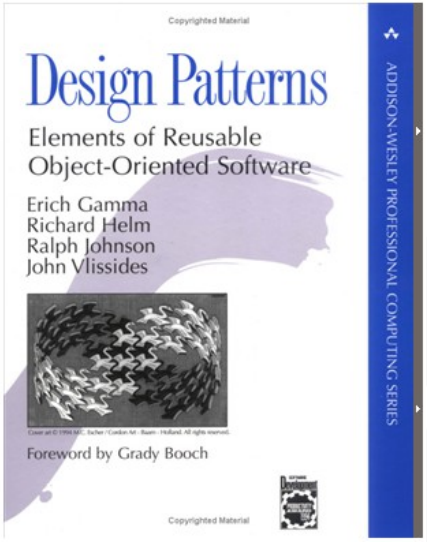
Descrição de um problema e de um pequeno conjunto de classes (estrutura e interação) que permitem resolver esse problema

- Documentam boas soluções para problemas recorrentes
  - Favorece o reuso dessas soluções
  - Reuso em termos de estruturas de classe

- Definem um vocabulário comum para conversar sobre projetos de software OO
  - Linguagem que introduz outro nível de abstração para lidar com sistemas complexos

Padrões de projeto

- A ideia de padrões foi apresentada pro Christopher Alexander em 1977 no contexto de arquitetura (de prédios e cidades)
- Outras ciências e engenharias possuem seus catálogos de padrões
- Desenvolvimento de software passou a ter seu primeiro catálogo em 1995
  - O livro da GoF (Gang of Four)
    - *E. Gamma and R. Helm and R. Johnson and J. Vlissides*
    - *Design Patterns – Elements of Reusable Object-Oriented Software*
    - *Addison-Wesley, 1995*



Estrutura de um Padrão

- Nome
- Problema resolvido por ele
- Como isso fornece uma solução
  - Estruturas, participantes, colaborações
- Consequencias
  - Resultados, compensações

Tipos de Padrões de Projeto

- Categorias de Padrões do GoF
- Padrões de Criação
  - Criação de objetos
- Padrões Estruturais
  - Composição de classes e objetos
- Padrões Comportamentais
  - Interação e distribuição de responsabilidades entre objetos e classes

Padrões de Projeto GoF

Classificação dos 23 padrões segundo GoF

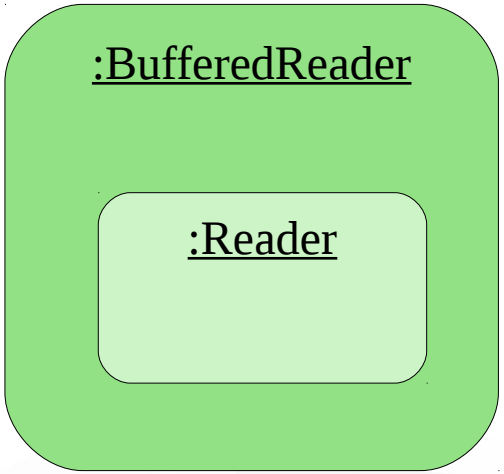
		Propósito		
		1. Criação	2. Estrutura	3. Comportamento
Escopo	Classe	Factory Method	Class Adapter	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Object Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Alguns padrões

- Decorator
- Singleton
- Factory
- Observer
- Adapter
- Template Method

Decorator

- Aumenta a funcionalidade de um objeto
- O objeto Decorator empacota outro objeto
  - Interfaces semelhantes
  - Chamada são recolocadas no objeto empacotado
  - Decorator pode interceptar e realizar ações adicionais
- Exemplos: java.io.BufferedReader
  - Empacota e aumenta um objeto Reader não-armazenado em buffer



Singleton

- Assegura a existência de uma única instância de uma classe
  - Todos os clientes usam o mesmo objeto
- O construtor é privado para impedir a instanciação externa
- Uma única instância obtida via um método getInstance estático
- Exemplo: Canvas no projeto *shapes*

```
//Exemplo de Singleton para a classe Parser
class Parser {
    private static Parser instance = new Parser();

    public static Parser getInstance()
    {
        return instance;
    }
    private Parser()
    {
        ...
    }
}
```

Fábrica (factory method)

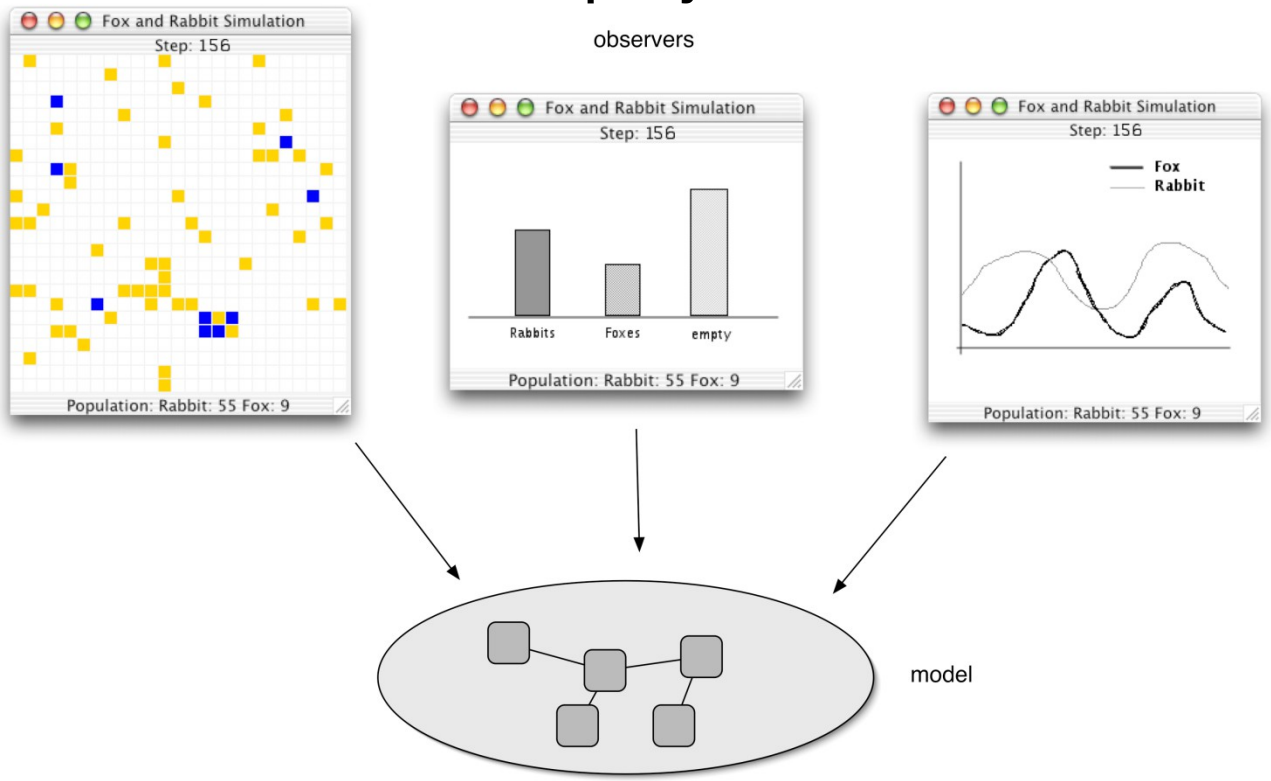
- Padrão criacional
- Clientes exigem um objeto de um tipo de interface particular ou tipo de superclasse
- Método de fábrica é livre para retornar um objeto de classe de implementação ou objeto de subclasse
- Tipo exato retornado depende do contexto
- Exemplo: método iterator da classe Collection

```
public void process(Collection<Type> coll)
{
    Iterator<Type> it = coll.iterator();
    ...
}
```

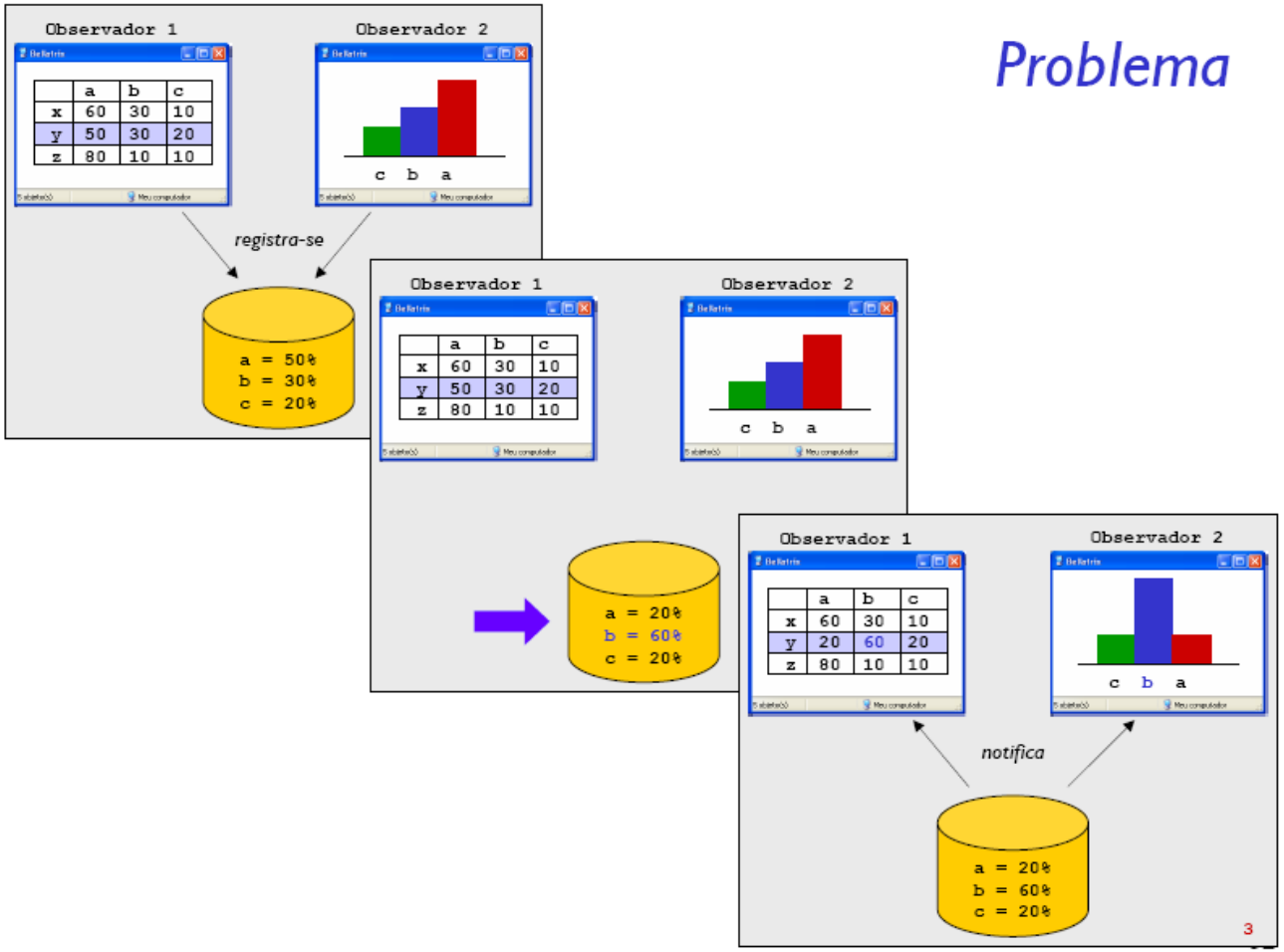
ArrayListIterator
HashSetIterator

Observer

- Suporta separação de modelo interno de uma visualização desse modelo
- Define relacionamento um-para-muitos entre objetos
- Objeto observado notifica todos os observadores de qualquer mudança de estado
- Exemplo: `SimulatorView` no projeto *foxes-and-rabbits*

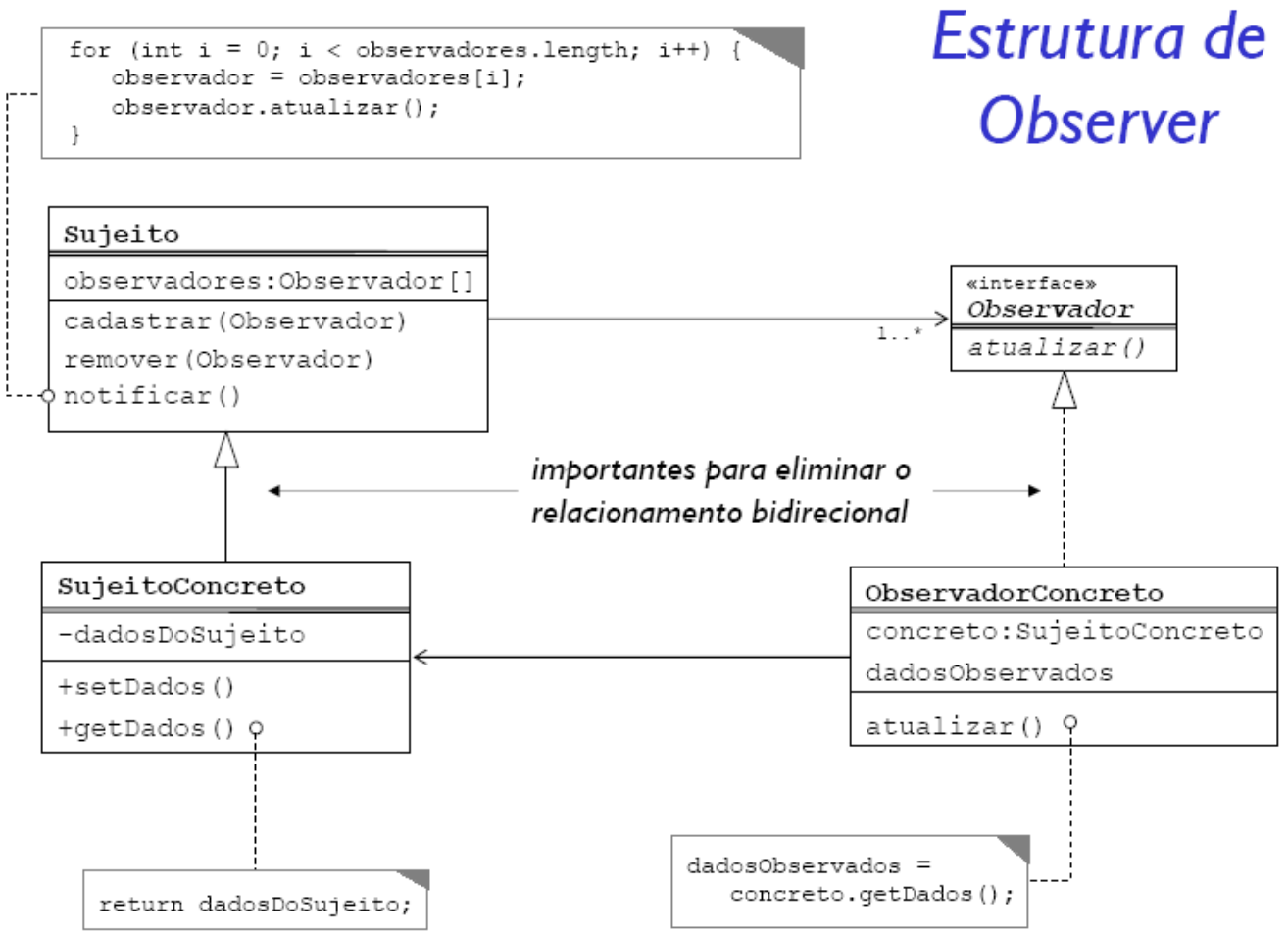


Observer



Problema

Observer


for (int i = 0; i < observadores.length; i++) {
observador = observadores[i];
observador.atualizar();
}
Estrutura de Observer
Sujeito
observadores:Observador[]
cadastrar(Observador)
remover(Observador)
notificar()
1..*
«interface»
Observador
atualizar()
importantes para eliminar o relacionamento bidirecional
SujeitoConcreto
-dadosDoSujeito
+setDados()
+getDados()
ObservadorConcreto
concreto:SujeitoConcreto
dadosObservados
atualizar()
return dadosDoSujeito;
dadosObservados = concreto.getDados();

Observer em Java

```
public class ConcreteObserver
             implements Observer {

  public void update(Observable o) {
    ObservableData data = (ObservableData) o;
    data.getData();
  }
}
```

```
public class ObservableData
             extends Observable {
  private Object myData;

  public void setData(Object myData) {
    this.myData = myData;
    notify();
  }

  public Object getData() {
    return myData();
  }
}
```

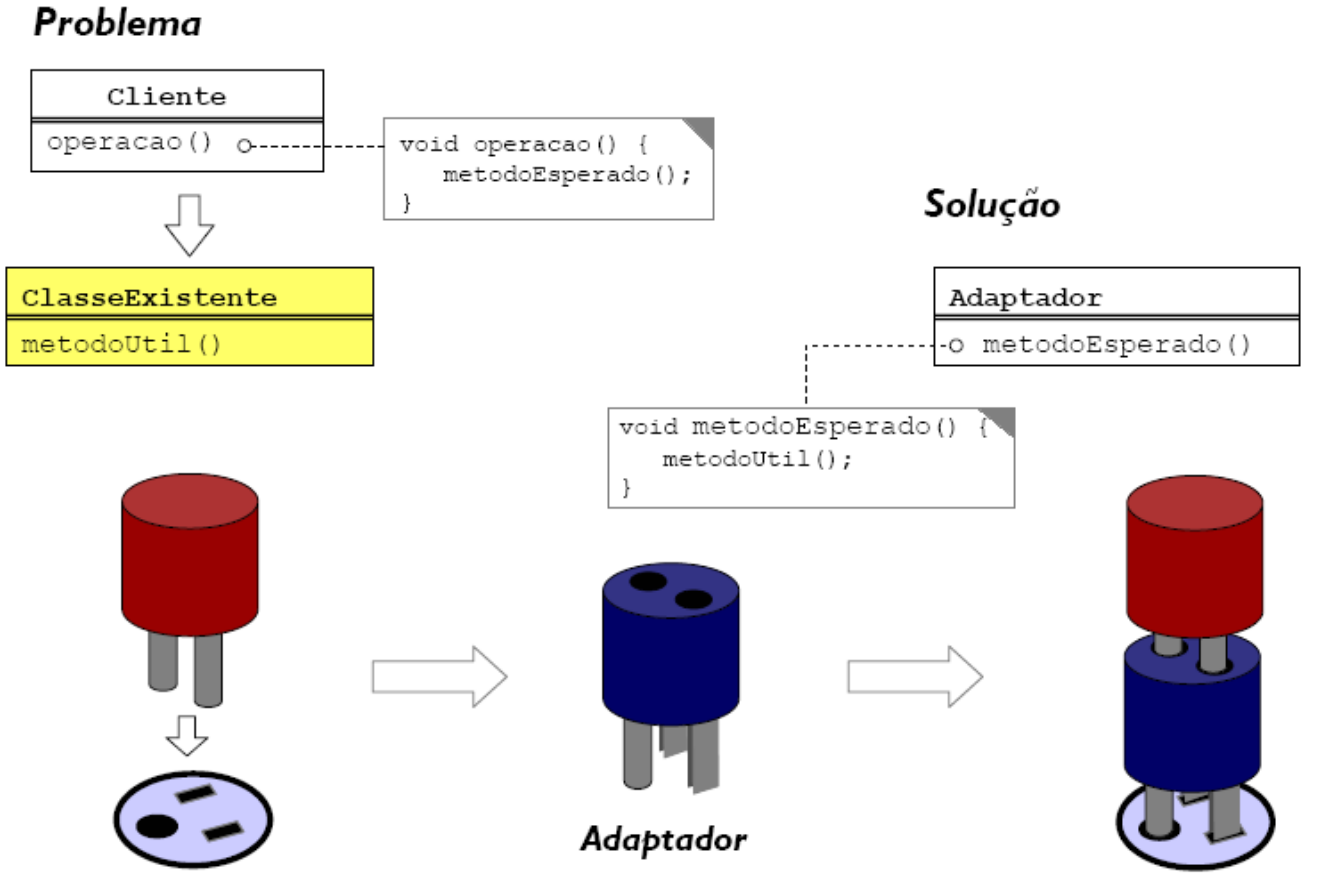
```
public class Observable {
  List observers = new ArrayList();

  public void add(Observer o) {
    observers.add(o);
  }

  public void remove(Observer o) {
    observers.remove(o);
  }

  public void notify() {
    Iterator it = observers.iterator();
    while(it.hasNext()) {
      Observer o = (Observer)it.next();
      o.update(this);
    }
  }
}
```

```
public interface Observer {
  public void update(Observable o);
}
```

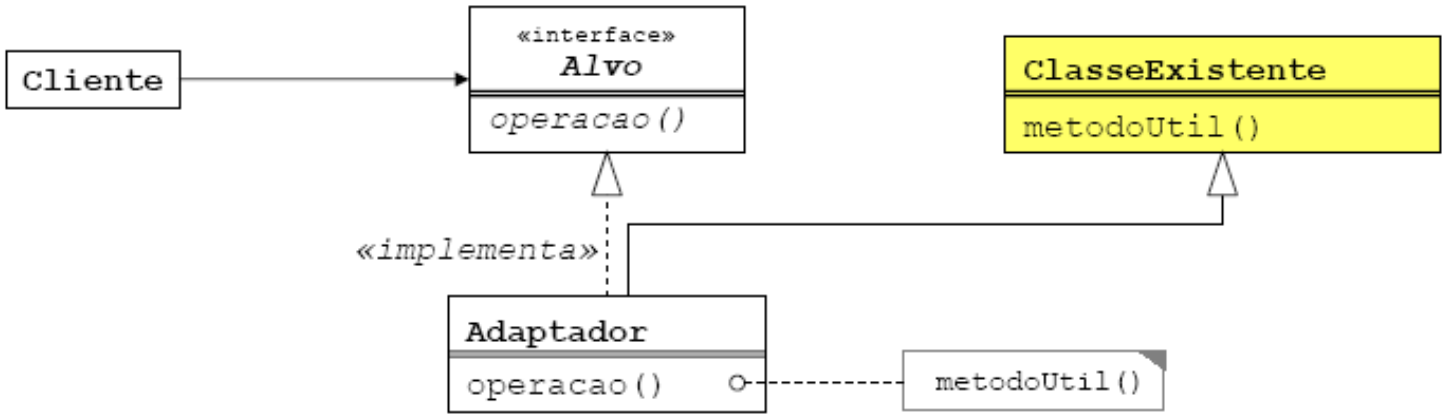
65

Adapter

- Converter a interface de uma classe em outra interface esperada pelos clientes
- Permite a comunicação entre classes que não poderiam trabalhar juntas devido à incompatibilidades de suas interfaces



Class Adapter



- *Cliente: aplicação que colabora com objetos aderentes à interface Alvo*
- *Alvo: define a interface requerida pelo Cliente*
- *ClasseExistente: interface que requer adaptação*

Class Adapter

Class Adapter em Java

```java
public class ClienteExemplo {
    Alvo[] alvos = new Alvo[10];
    public void inicializaAlvos() {
        alvos[0] = new AlvoExistente();
        alvos[1] = new Adaptador();
        // ...
    }
    public void executaAlvos() {
        for (int i = 0; i < alvos.length; i++) {
            alvo.operacao();
        }
    }
}
```

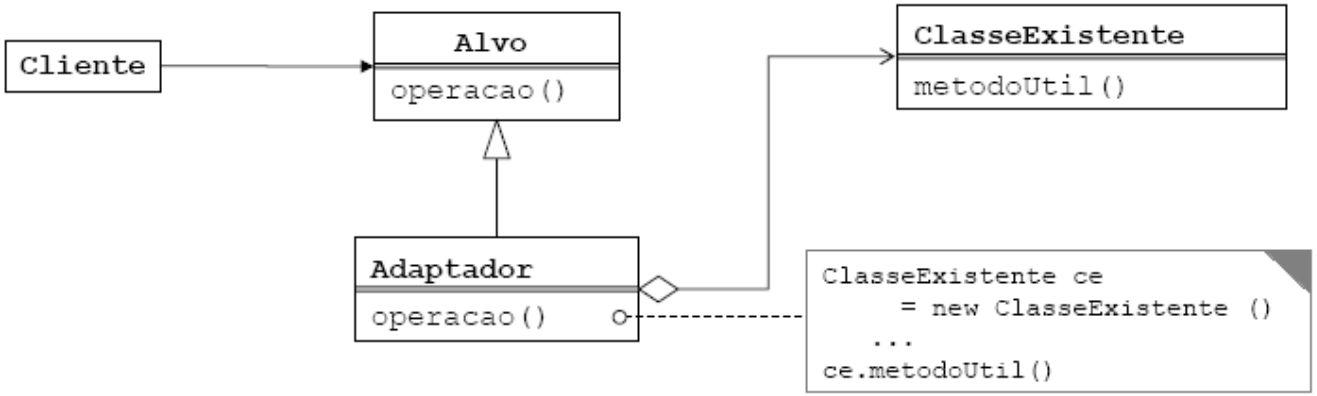
```java
public interface Alvo {
    void operacao();
}
```

```java
public class Adaptador extends ClasseExistente implements Alvo {
    public void operacao() {
        String texto = metodoUtilDois("Operação Realizada.");
        metodoUtilUm(texto);
    }
}
```

```java
public class ClasseExistente {
    public void metodoUtilUm(String texto) {
        System.out.println(texto);
    }
    public String metodoUtilDois(String texto) {
        return texto.toUpperCase();
    }
}
```

24

Object Adapter



- *Única solução se Alvo não for uma interface Java*
- *Adaptador possui referência para objeto que terá sua interface adaptada (instância de ClasseExistente).*
- *Cada método de Alvo chama o(s) método(s) correspondente(s) na interface adaptada.*

Object Adapter

Object Adapter em Java

```java
public class ClienteExemplo {
    Alvo[] alvos = new Alvo[10];
    public void inicializaAlvos() {
        alvos[0] = new AlvoExistente();
        alvos[1] = new Adaptador();
        // ...
    }
    public void executaAlvos() {
        for (int i = 0; i < alvos.length; i++) {
            alvos[i].operacao();
        }
    }
}
```

```java
public abstract class Alvo {
    public abstract void operacao();
    // ... resto da classe
}
```

```java
public class Adaptador extends Alvo {
    ClasseExistente existente = new ClasseExistente();
    public void operacao() {
        String texto = existente.metodoUtilDois("Operação Realizada.");
        existente.metodoUtilUm(texto);
    }
}
```

```java
public class ClasseExistente {
    public void metodoUtilUm(String texto) {
        System.out.println(texto);
    }
    public String metodoUtilDois(String texto) {
        return texto.toUpperCase();
    }
}
```

25

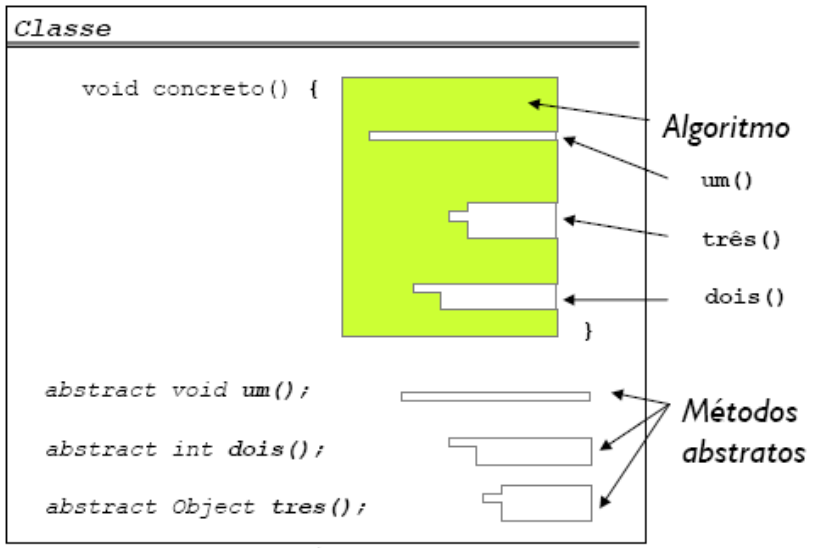
Adapter

- Quando usar
  - Sempre que for necessário adaptar uma interface para um cliente
- Class Adapter
  - Quando houver uma interface que permita a implementação estática
- Object Adapter
  - Quando um menor acoplamento for desejado
  - Quando o cliente não usa uma interface ou classe abstrata

Template Method

- Definir o esqueleto de um algoritmo dentro de uma operação, deixando alguns passos a serem preenchidos pelas subclasses
- Subclasses redefinem certos passos de um algoritmo sem mudar sua estrutura

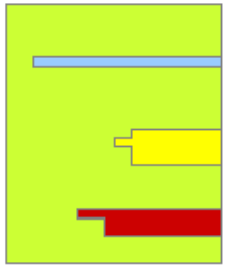
Template Method



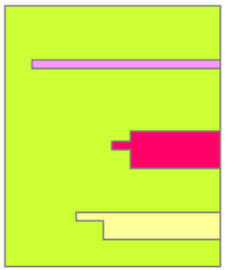
Problema

Algoritmos resultantes

```
Classe x =
  new ClasseConcretaUm()
x.concreto()
```



```
Classe x =
  new ClasseConcretaDois()
x.concreto()
```



127

Template Method

- Define um algoritmo em termos de operações abstratas
- Operações são implementadas por subclasses para oferecer comportamento concreto
- Quando usar
  - Quando a estrutura fixa de um algoritmo puder ser definida pela superclasse deixando certas partes para serem preenchidas por implementações que podem varia

Template Method

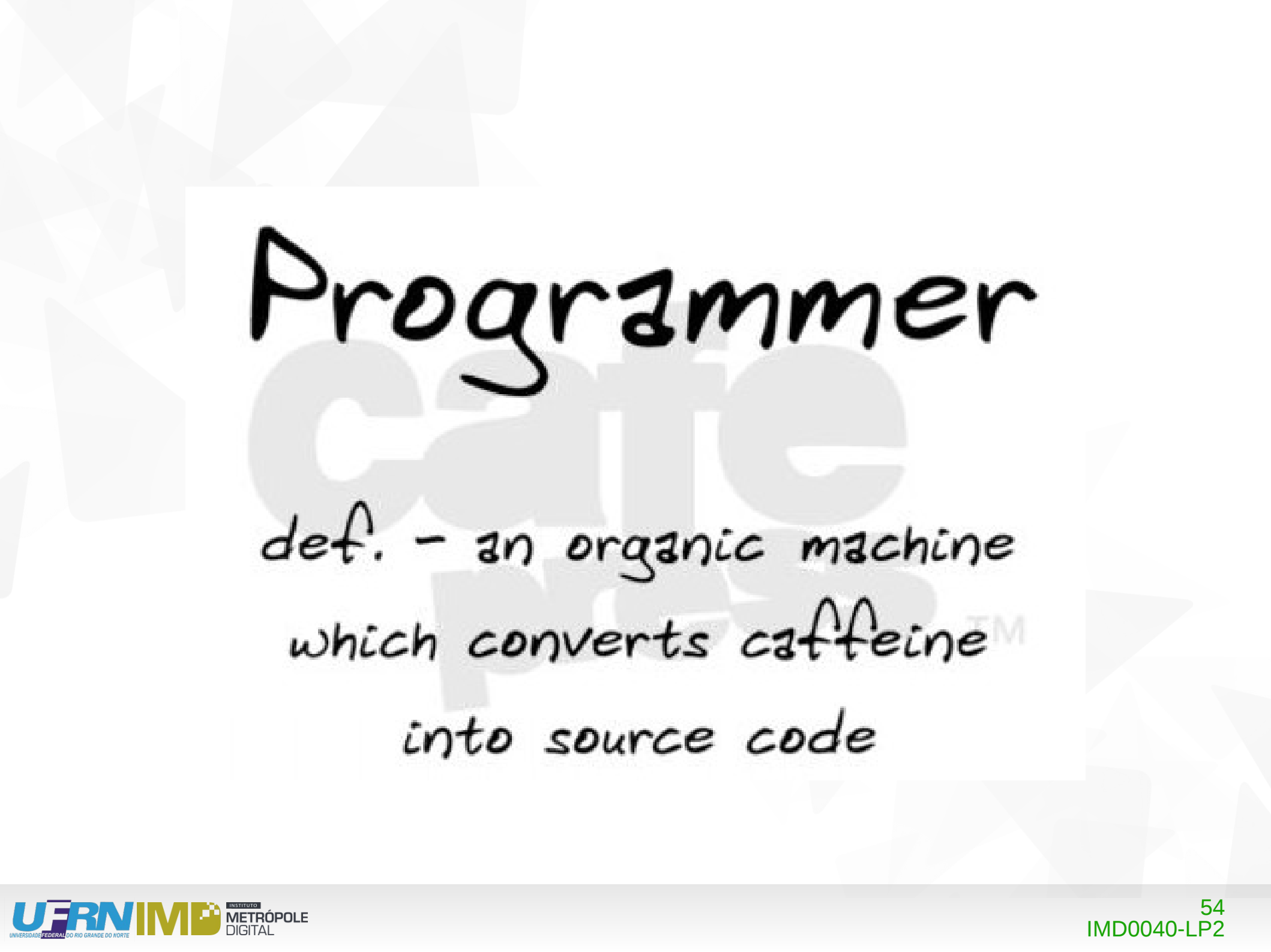
```
public abstract class Template {
   protected abstract String link(String texto, String url);
   protected String transform(String texto) { return texto; }
   public final String templateMethod() {
      String msg = "Endereço: " + link("Empresa", "http://www.empresa.com");
      return transform(msg);
   }
```

```
public class XMLData extends Template {
   protected String link(String texto, String url) {
      return "<endereco xlink:href='"+url+"'>"+texto+"</endereco>";
   }
}
```

```
public class HTMLData extends Template {
   protected String link(String texto, String url) {
      return "<a href='"+url+"'>"+texto+"</a>";
   }
   protected String transform(String texto) {
      return texto.toLowerCase();
   }
```

Revisão

- Colaborações de classe e interações de objeto devem ser identificadas
  - Análise CRC suporta isso
- Uma abordagem iterativa ao design, análise e implementação pode ser benéfica
  - Considere os sistemas de software como entidades que crescerão e se desenvolverão ao longo do tempo
- Trabalhe de forma que facilite a colaboração com outras pessoas
- Projete estrutura de classe flexíveis e extensíveis
  - Ficar atento aos padrões de projeto existentes ajudará você a fazer isso


Programmer
def. - an organic machine
which converts caffeine
into source code

Exercícios

- Exercício 13.7 – Realize a análise e projeto da descrição abaixo:
-

O programa é um sistema de simulação de voos. Para nosso novo aeroporto precisamos saber se podemos operar com duas pistas ou se precisamos de três O aeroporto funciona assim: O aeroporto tem varias pistas. Os aviões decolam e aterrissam nas pistas. Os controladores do tráfego aéreo coordenam o tráfego e dão permissão para decolagem e aterrissagem. Os controladores às vezes dão permissão imediata, outras vezes eles informam que os aviões precisam esperar. Os aviões devem manter certa distância entre eles. O propósito do programa é simular o aeroporto em operação

Por hoje é só....

- Capítulo 13 do livro

IMD0040
Linguagem de Programação 2

- Aula 22
- Design de Aplicações