Instituto de Computação

Design Patterns

Viviane Torres da Silva
viviane.silva@ic.uff.br

http://www.ic.uff.br/~viviane.silva/2010.1/es1

Sumário

- Reuso de Software
  - Introdução
  - Benefícios e Desvantagens
  - Visão do Reuso
- Padrões de Projeto
  - Introdução
  - Motivação
  - Alguns Padrões
    - Singleton
    - Facade
    - Command
    - Observer
    - DAO
    - Filter

Reuso de Software

Introdução

- Maioria das Engenharias
  - Desenvolvimento de sistemas
    - Composição de componentes existentes
    - Componentes usados em outros sistemas
- Engenharia de Software
  - Antes
    - Focado no desenvolvimento original
  - Agora
    - Processo de desenvolvimento baseado em um reuso de software sistematizado, trazendo
      - Software de melhor qualidade
      - Desenvolvimento mais rápido
      - Menor custo

Engenharia de Software baseada em Reuso

- Reuso de Sistemas
  - Incorporação de um sistema, sem alterá-lo, em outro sistema (COTS)
  - Desenvolvimento de famílias de aplicações
- Reuso de Componentes
  - Sub-sistemas de uma aplicação a simples objetos
- Reuso de Objetos e Funções
  - Objetos simples e bem definidos
  - Funções

Benefícios

- Confiabilidade Crescente
  - Toda vez que um software é utilizado, ele é novamente testado
  - Componentes já utilizados e testados em outros sistemas são mais confiáveis que novos componentes
- Risco de Processo Reduzido
  - Margem de erro dos custos de reuso menor que dos custos de desenvolvimento
- Uso Efetivo de Especialistas
  - Especialista desenvolve software reutilizável encapsulando seu conhecimento, ao invés de desenvolver as mesmas funcionalidades repetidas vezes em diferentes projetos

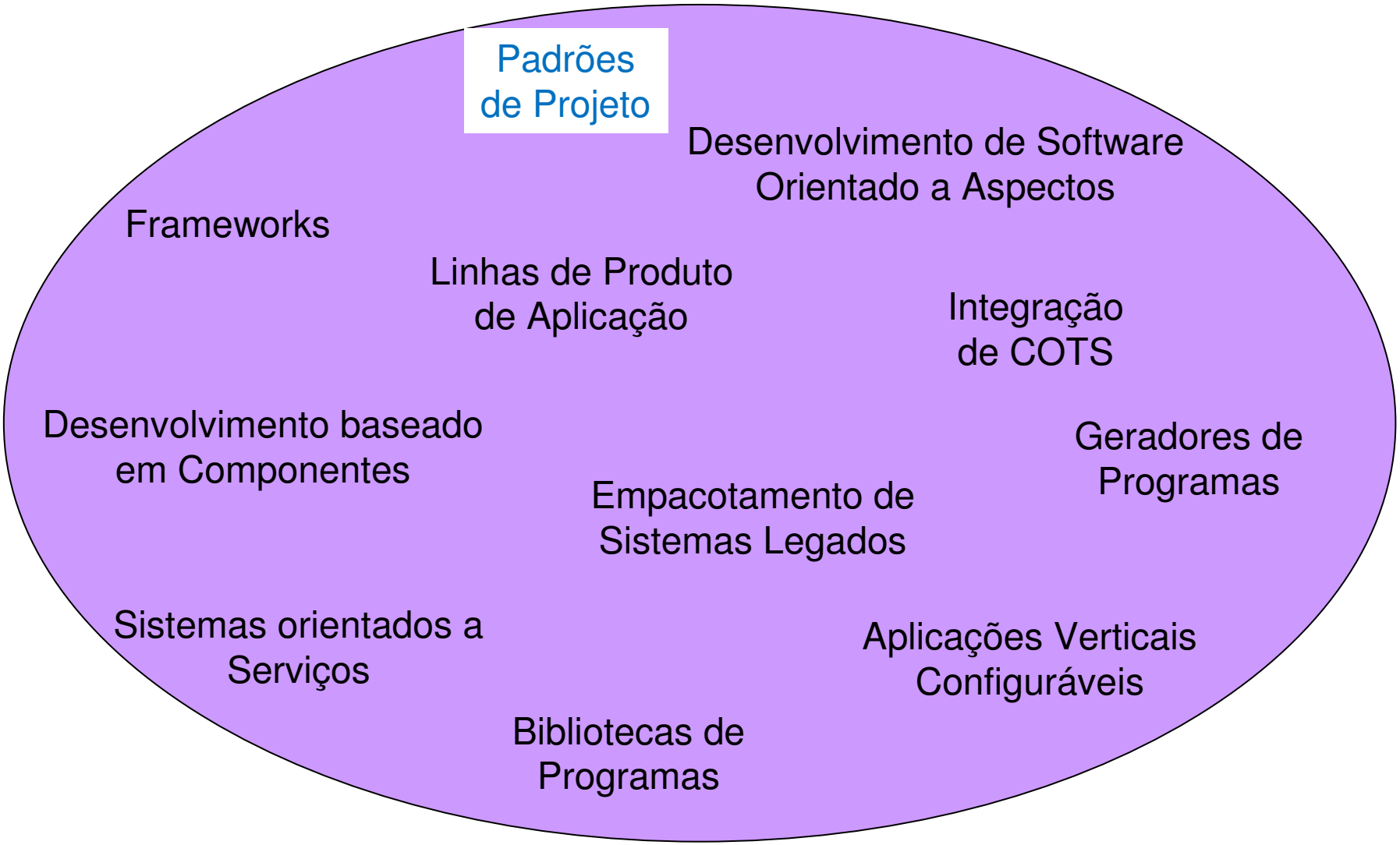
Benefícios

- Conformidade com Padrões
  - Uso de padrões organizacionais agiliza o desenvolvimento
    - Estabelece uma base comum de comunicação
    - Garante a consistência
  - Exemplo: padrões de interface
- Desenvolvimento Acelerado
  - Redução do tempo de desenvolvimento e de validação

Problemas

- Custos de Manutenção Crescente
  - Dificuldade de adaptar componentes sem o código fonte
- Falta de Ferramentas de Suporte
  - Ferramentas CASE podem não suportar desenvolvimento com reuso
- Síndrome do "não foi inventado aqui"
  - Falta de confiança no componente
  - Desenvolver é visto como mais desafiador que reutilizar
- Criar e Manter um biblioteca de Componentes
  - Custo de criar e manter a biblioteca pode ser grande
  - Técnicas de classificar, catalogar e recuperar os componentes são imaturas
- Encontrar, Entender e Adaptar Componentes Reusáveis
  - Busca de componentes como parte do processo de desenvolvimento

Visão do Reuso



Visão do Reuso

- Padrões de Projeto
  - Abstrações genéricas que ocorrem nas aplicações
- Desenvolvimento baseado em Componentes
  - Sistemas desenvolvidos pela integração de componentes
- Frameworks
  - Coleção de classes abstratas e concretas que podem ser adaptadas e estendidas para a criação de aplicações
- Empacotamento de Sistemas Legados
  - Interfaces podem ser definidas para prover acesso a sistemas legados
- Sistemas Orientados a Serviços
  - Sistemas desenvolvidos pela ligação com serviços compartilhados
  - Serviços podem ser externos

Visão do Reuso

- Linhas de Produto de Aplicação
  - Tipo de aplicação generalizado em uma arquitetura comum que pode ser adaptada de diferentes modos para diferentes clientes
- Integração de COTS (*Commercial off-the-shelf*)
  - Termo que permite desenvolver a partir de componentes já criados e realizar adaptações
  - Sistemas desenvolvidos pela integração de aplicações existentes
- Aplicações Verticais Configuráveis
  - Desenvolvimento de sistemas genéricos que podem ser configurados às necessidades de clientes de um sistema específico
- Bibliotecas de Programas
  - Biblioteca de Classes e Funções comumente usadas
- Geradores de Programas
  - Sistema Gerador tem conhecimento de tipos particulares de aplicação e pode gerar sistemas ou fragmentos de sistemas
- Desenvolvimento de Software Orientado a Aspectos
  - Componentes compartilhados são entrelaçados na aplicação em diferentes partes quando o programa é compilado

Padrões de Projeto

Definição: Padrão

"Cada **padrão** descreve um **problema** que ocorre repetidas vezes em nosso **ambiente**, e então descreve o núcleo da sua **solução** para aquele **problema**, de tal maneira que seja possível usar essa solução milhões de vezes sem nunca fazê-la da mesma forma duas vezes."

Christopher Alexander sobre padrões em arquitetura de construções

Definição: Padrão de Projeto

"Os padrões de projeto são **descrições de objetos** que se comunicam e classes que são customizadas para resolver um **problema** de projeto genérico em um **contexto** específico."

Gamma, Helm, Vlissides & Johnson, sobre padrões de projeto em software

Definição: Padrão de Projeto

- Forma de reusar conhecimento abstrato sobre um problema e sua solução

- Suficientemente abstrato para ser reusado sob diferentes contextos
  - descrições de problemas e essências de soluções
  - aplicáveis em classes de problemas bem conhecidos
  - soluções que funcionam, tornando-se "receitas" para situações similares

- Freqüentemente usa características da OO como herança e polimorfismo

Definição: Padrão de Projeto

- Inspirados em "A Pattern Language" de Christopher Alexander
  - Padrões de arquitetura de cidades, casas e prédios

- Design Patterns: Elements of Reusable Object-Oriented Software
  - Catálogo publicado em 1994
  - Erich Gamma, John Vlissides, Ralph Jonhson e Richard Helm, conhecidos como "The Gang of Four" (GoF)
  - 23 padrões de projeto

Benefícios

- Aprendizagem com a experiência dos outros
  - Identificação de problemas comuns de projeto de software
  - Utilização de soluções testadas e bem documentadas
  - Ajuda um novato a agir mais como um experiente
- Produção de bons projetos orientados a objetos
  - Normalmente utilizam boas práticas de OO
  - Utilizam eficientemente polimorfismo, herança e composição
- Vocabulário comum
  - Uso de soluções que têm nome facilita comunicação
  - Nível mais alto de abstração
- Ajuda na documentação
  - Uso de soluções que têm um nome facilita a documentação
  - Conhecimento de padrões de projeto torna mais fácil a compreensão de sistemas existentes
- Ajuda na conversão de um modelo de análise em um modelo de implementação
- Aumento da produtividade

Elementos Essenciais

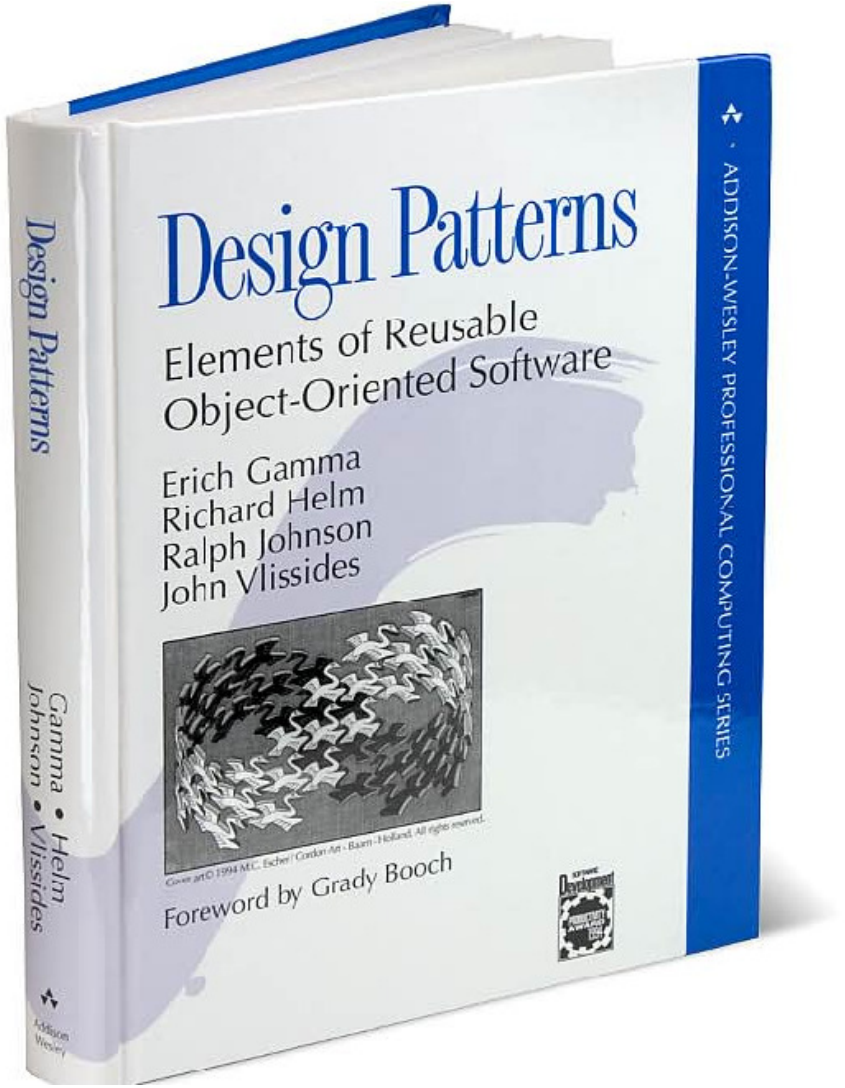
- Nome
  - Procura descrever o problema, a solução e as conseqüências em uma ou duas palavras.
- Problema
  - Quando aplicar o padrão e em que condições
- Solução
  - Descrição abstrata de um problema
  - Como usar os elementos disponíveis (classes e objetos) para solucioná-lo
- Conseqüências
  - Custos e benefícios de se aplicar o padrão
  - Impacto na flexibilidade, reusabilidade e eficiência do sistema

Livros sobre Padrões de Software

Categoria do Padrão	Título	Autores / Editores
Análise OO	Analysis Patterns: Reusable Object Models	Martin Fowler
Arquitetura	**Pattern-Oriented Software Architecture: A System of Patterns**	**Buschmann et al.**
Projeto	**Design Patterns: Elements of Reusable Object-Oriented Software**	**Gamma et al.**
	Anti-Patterns: Refactoring Software, Architectures, and Projects in Crisis	William J. Brown et al.
	Design Patterns Java™ Workbook	Steven John Metsker

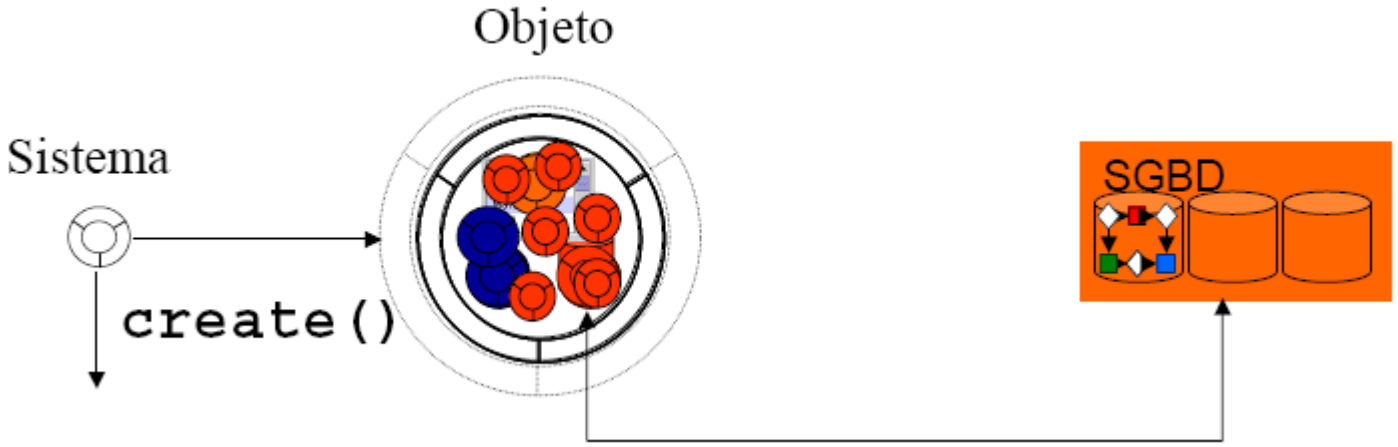
Livro: Padrões de Projeto do GoF

- Catálogo de 23 padrões
- Não apresenta padrão para um domínio de aplicação específico
- Padrões do GoF representam o estado-da-prática em boas construções de projeto orientado a objetos
- É comum encontrar no detalhamento de padrões específicos de domínio a ocorrência de algum dos padrões do GoF



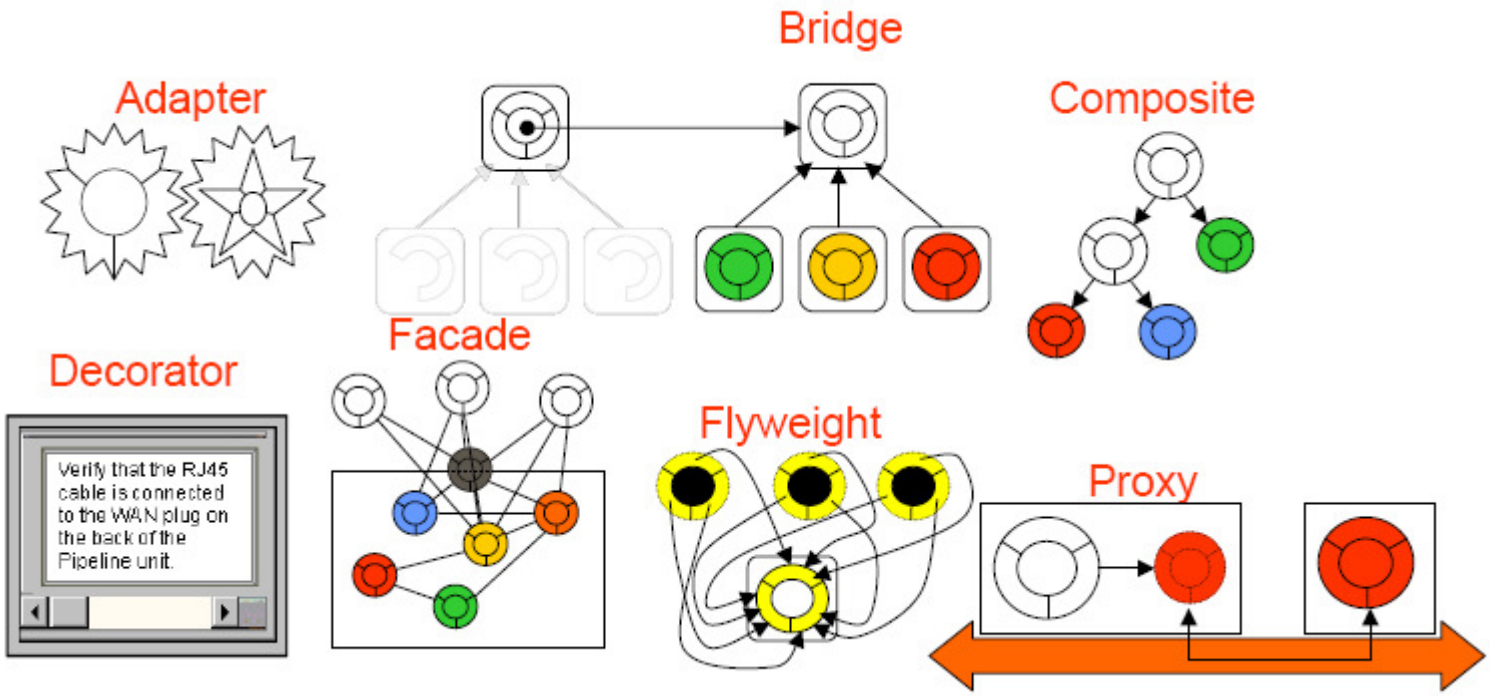
Livro: Padrões de Projeto da GoF

- Classificação
  - Padrões de Criação
    - Abstraem o processo de instanciação
    - Tornam um sistema independente da forma como os objetos são criados, compostos e representados



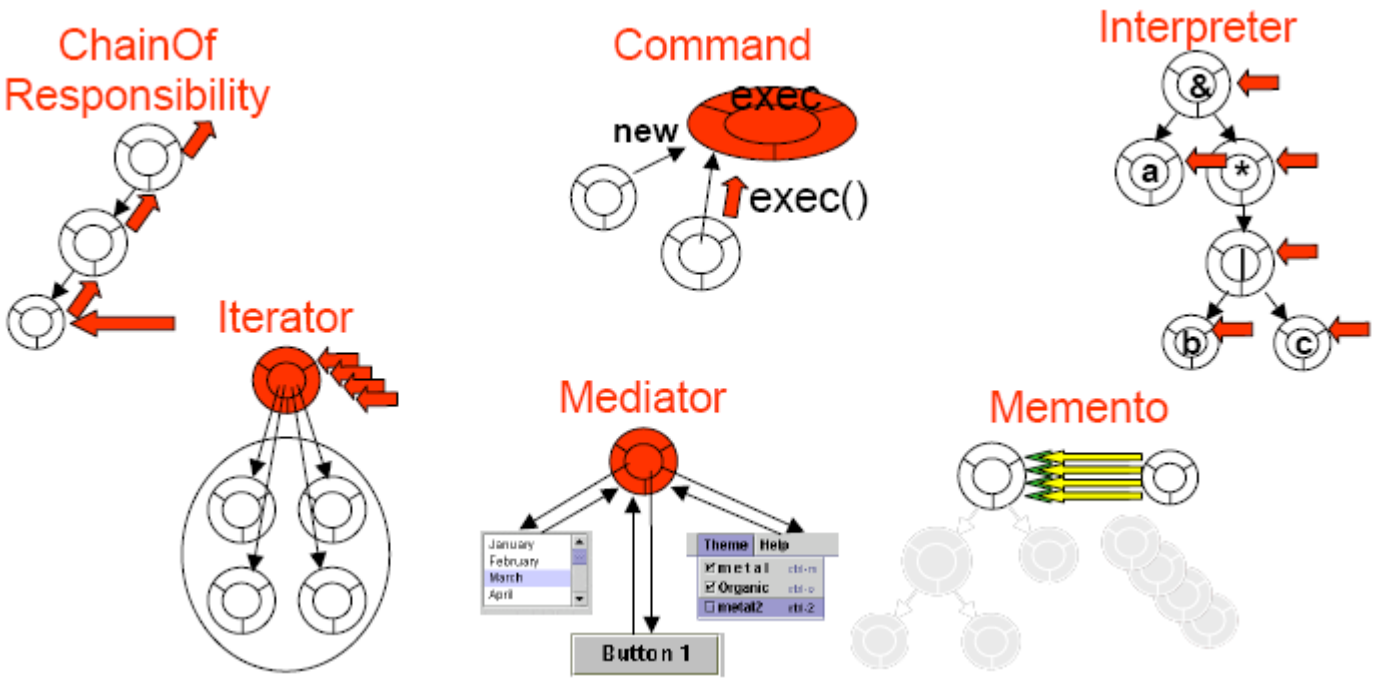
Livro: Padrões de Projeto da GoF

- Classificação
  - Padrões Estruturais
    - Lidam com a composição de classes (ou objetos) para formar grandes estruturas no sistema



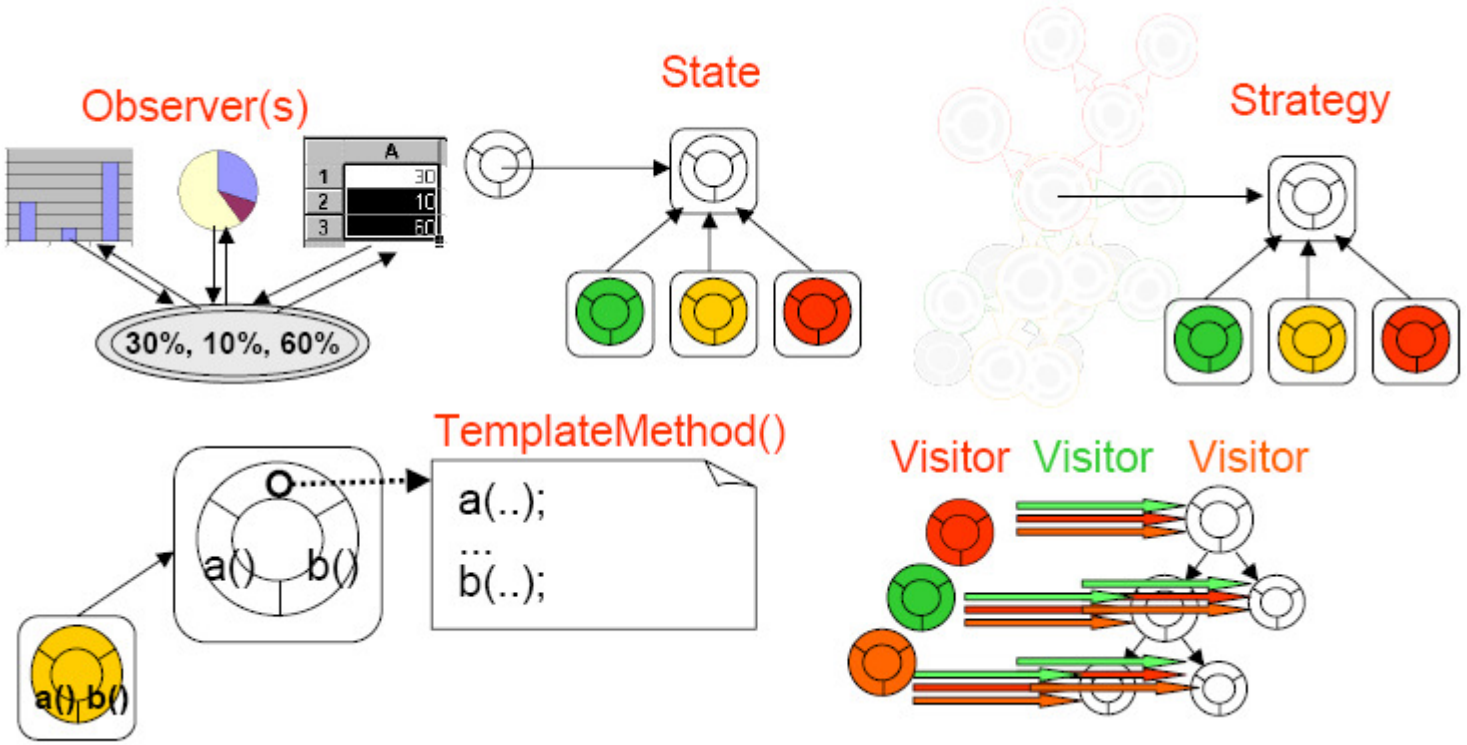
Livro: Padrões de Projeto da GoF

- Classificação
  - Padrões Comportamentais
    - Caracterizam a forma como classes (ou objetos) interagem
    - Distribuem responsabilidade



Livro: Padrões de Projeto da GoF

- Classificação
  - Padrões Comportamentais
    - Caracterizam a forma como classes (ou objetos) interagem
    - Distribuem responsabilidade



Livro: Padrões de Projeto da GoF

1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton
6. Adapter
7. Bridge
8. Composite
9. Decorator
10. Facade
11. Flyweight
12. Proxy
13. Chain of Responsibility
14. Command
15. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template Method
23. Visitor

- Padrões de Criação
- Padrões Estruturais
- Padrões de Comportamento

Livro: Padrões de Projeto da GoF

- Template
  1. Pattern Name and Classification
  2. Intent
  3. Also Known as
  4. Motivation
  5. Applicability
  6. Structure
  7. Participants
  8. Collaborations
  9. Consequences
  10. Implementation
  11. Sample Code
  12. Known Uses
  13. Related Patterns

Alguns Padrões de Projeto

Alguns Padrões de Projeto

- GoF
  - Singleton (criação)
  - Façade (estrutural)
  - Template Method (comportamental)
  - Strategy (comportamental)
  - State (comportamental)
  - Observer (comportamental)

- Outros padrões
  - DAO

Singleton

- Motivação
  - Garantir que exista um determinado número X de objetos de uma classe
    - Independentemente do número de requisições que receber para criá-lo
  - Exemplos de aplicação
    - Único banco de dados
    - Único acesso ao arquivo de log
    - Única fachada (padrão Facade)

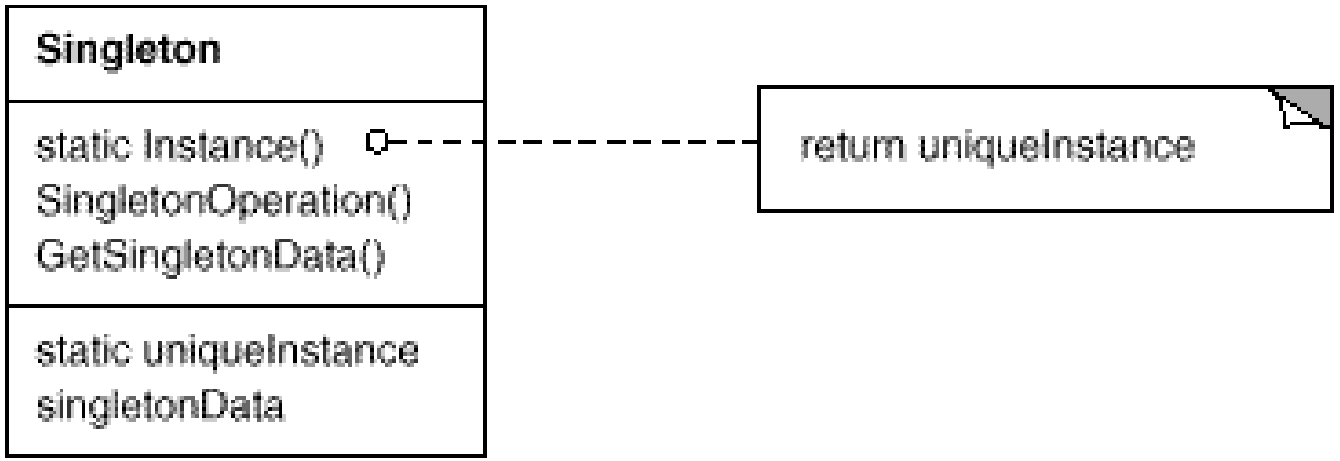
Singleton

- Propósito
  - Assegurar o controle da quantidade de instâncias da classe
  - Ponto de acesso global a ela

- Aplicabilidade
  - Exatamente uma instância da classe
    - Acessível pelos clientes de ponto de acesso bem conhecido
  - Instância única deve ser extensível através de subclasses
    - Clientes capazes de usar instância estendida sem alterar seu código

Singleton

- Estrutura



- Participantes
  - Singleton
    - Define operação **Instance** que permite que clientes acessem instância única
      - Instance é operação de classe
    - Pode ser responsável pela criação de sua única instância

Singleton

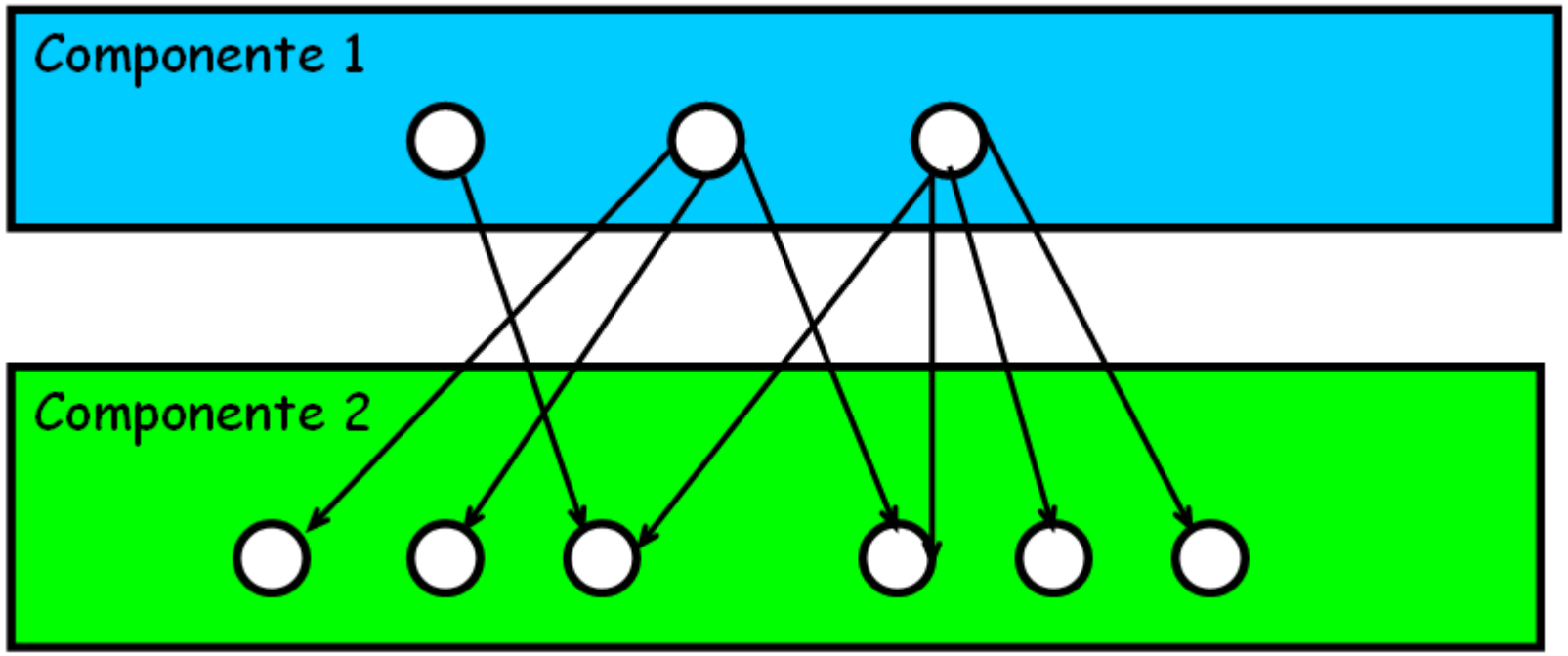
- Conseqüências
  - Acesso controlado a instância única
  - Espaço de nomes reduzido
  - Refinamento de operações e representação
  - Não há número variado de instâncias
  - Mais flexível do que operações de classes

Singleton

```
public class Singleton  {
    private static Singleton instance = null;

    public synchronized static Singleton getInstance(){
        if  (instance == null) {
                instance = new Singleton();
        }
        return instance;
    }
    private Singleton () {
    }
}
```

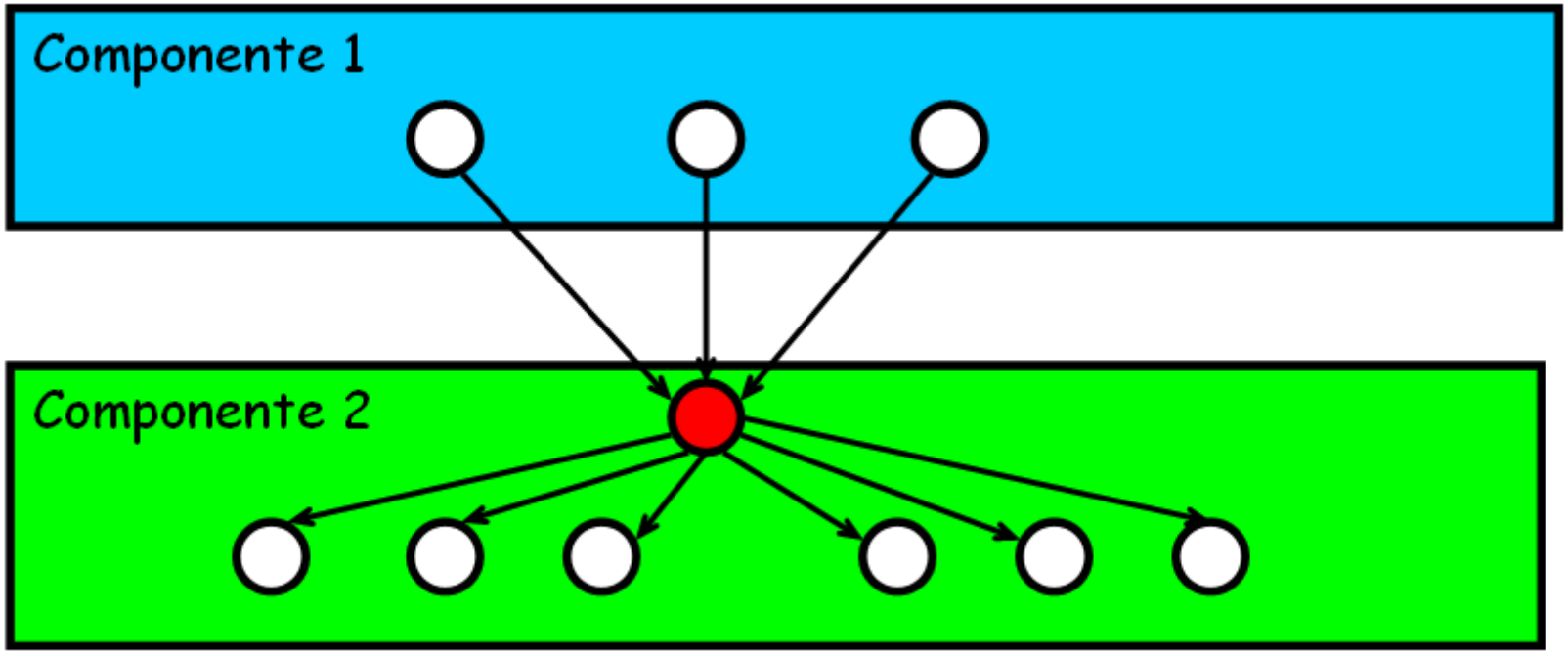
Facade

- Motivação



Facade

- Motivação



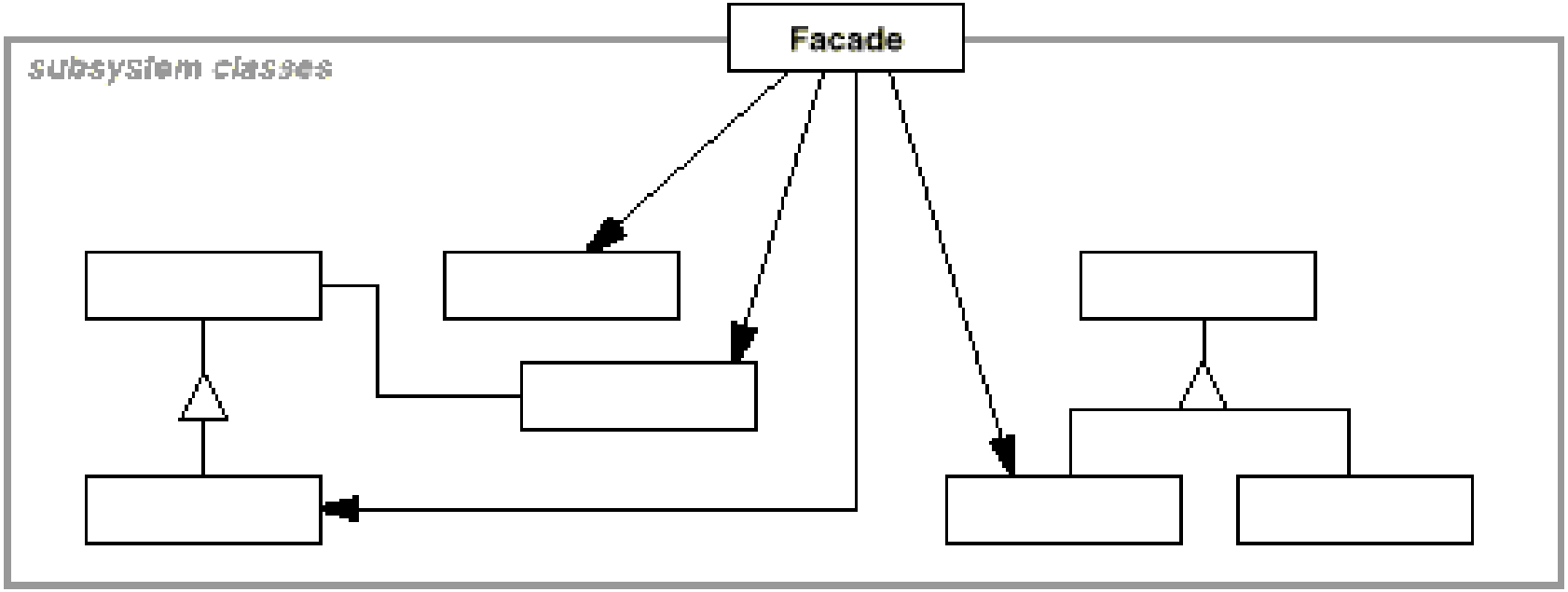
Facade

- Propósito
  - Prover interface unificada para conjunto de interfaces em um subsistema
  - Define interface de alto-nível
    - Subsistema mais fácil de usar

- Aplicabilidade
  - Prover interface simples para subsistema complexo
  - Muitas dependências entre clientes e classes que implementam uma abstração
  - Criar camadas no subsistema

Facade

- Estrutura



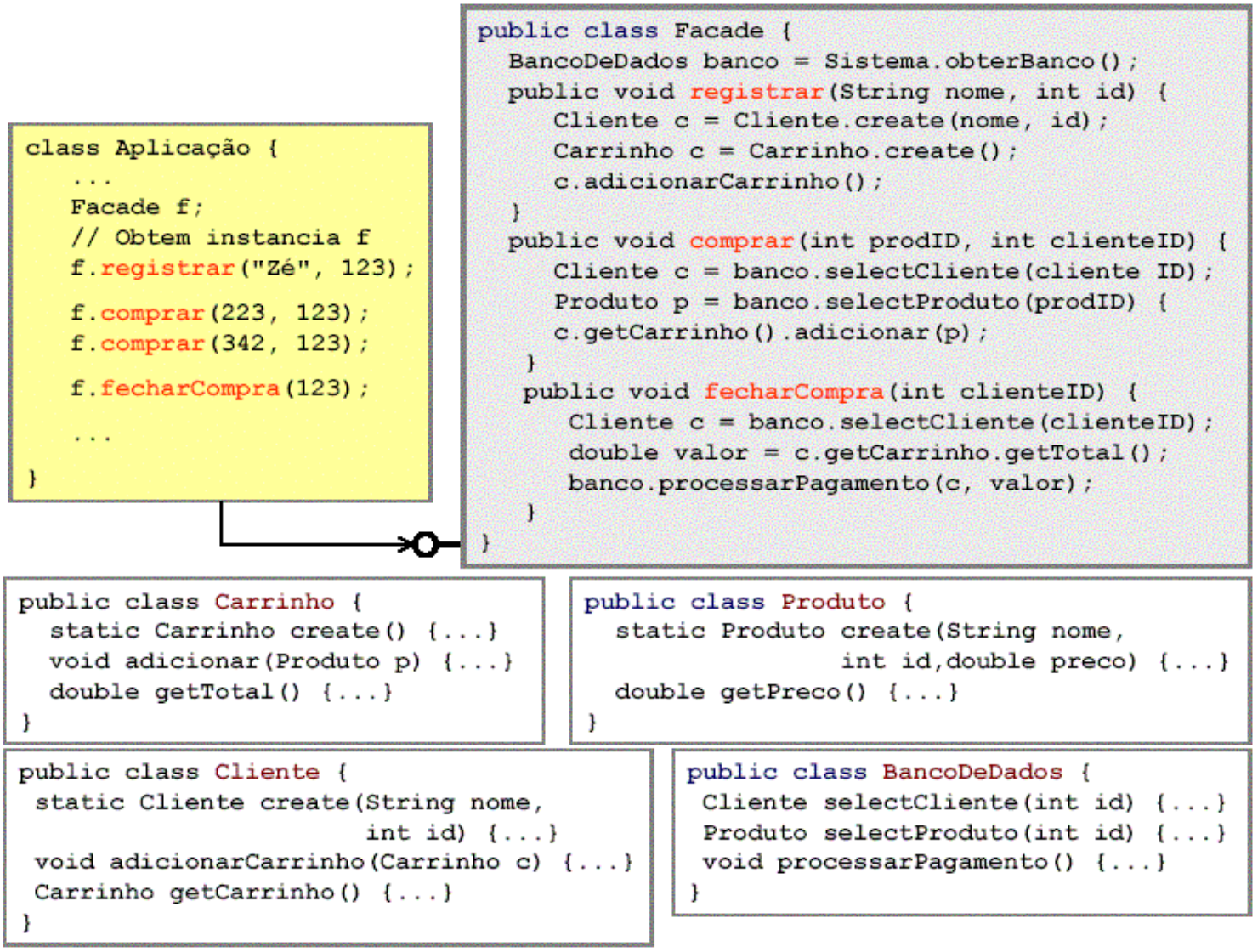
Facade

- Participantes
  - Facade
    - Conhece quais classes do subsistema seriam responsáveis pelo atendimento de uma solicitação
    - Delega solicitações de clientes a objetos apropriados do subsistemas
  - Classes de subsistema
    - Implementam as funcionalidades do subsistema
    - Respondem a solicitações de serviços da Facade
    - Não têm conhecimento da Facade

Facade

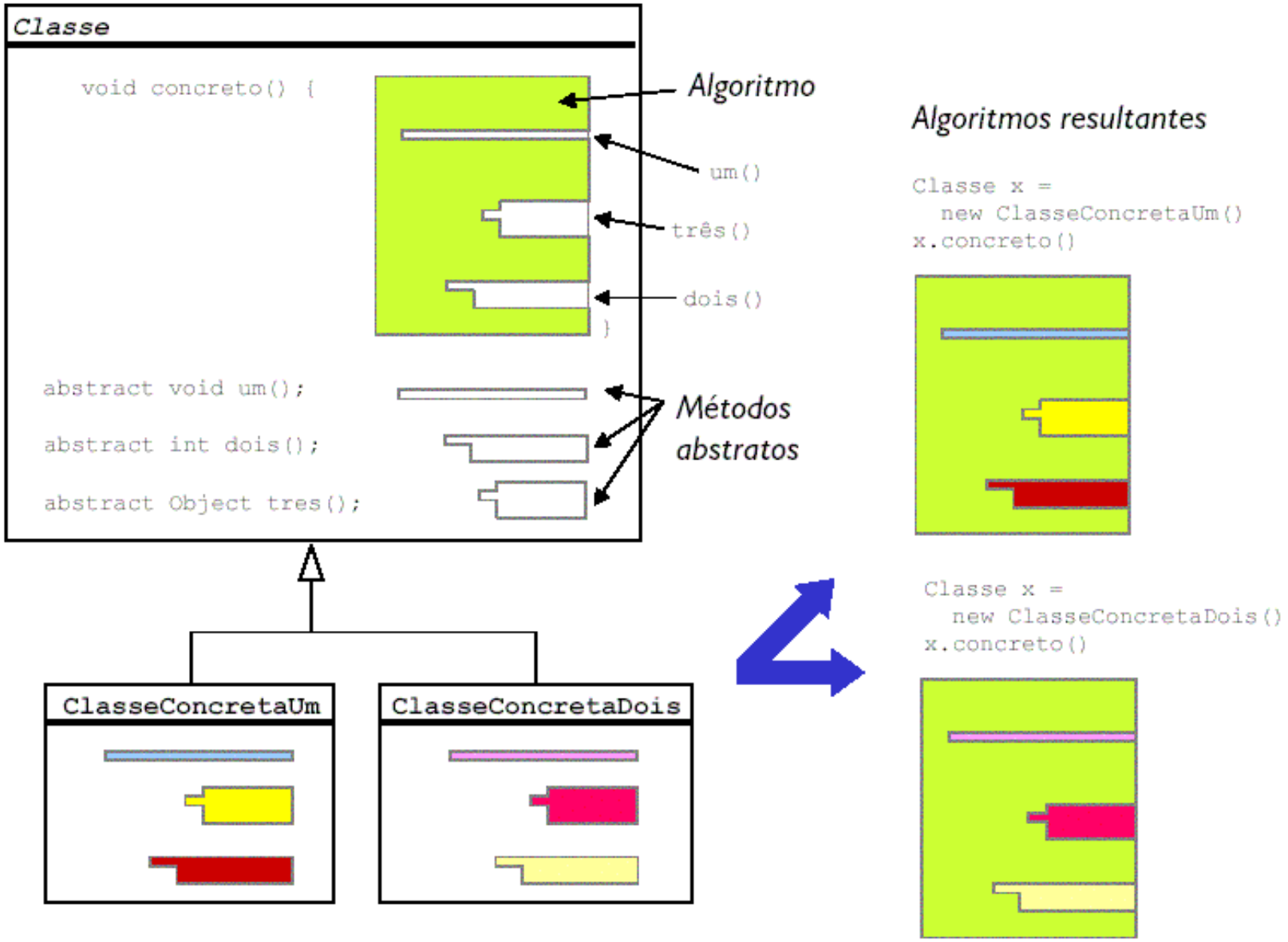
- Conseqüências
  - Esconde do cliente os componentes do subsistema
    - Reduz o número de objetos que os clientes lidam
    - Subsistema mais fácil de usar
  - Fraco acoplamento entre subsistema e seus clientes
  - Não impede que aplicações usem classes do subsistema, caso elas precisem

Facade


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() {...}
}

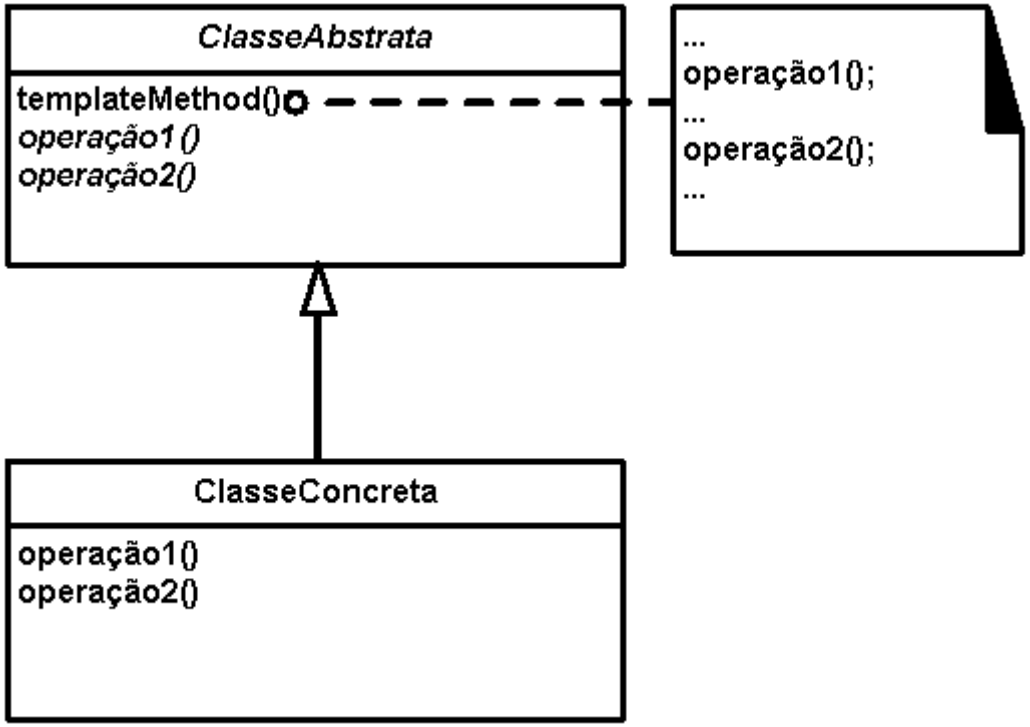
Template Method

- Motivação



Template Method

- Estrutura



Template Method

- Objetivo
  - Define o esqueleto de um algoritmo numa operação, deixando que subclasses completem algumas das etapas
  - O padrão Template Method permite que subclasses redefinem determinadas etapas de um algoritmo sem alterar a estrutura do algoritmo

Template Method

- Participantes
  - ClasseAbstrata (Login):
    - Define operações abstratas que subclasses concretas definem para implementar certas etapas do algoritmo
    - Implementa um Template Method definindo o esqueleto de um algoritmo
    - O Template Method chama várias operações, entre as quais as operações abstratas da classe
  - ClasseConcreta (LoginDecisionSupportSystem) :
    - implementa as operações abstratas para desempenhar as etapas do algoritmo que tenham comportamento específico a esta subclasse

Template Method

- Conseqüência:
  - Template Methods constituem uma das técnicas básicas de reuso de código
  - Template Methods levam a uma inversão de controle

Template Method

- Exemplo de Código

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

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

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

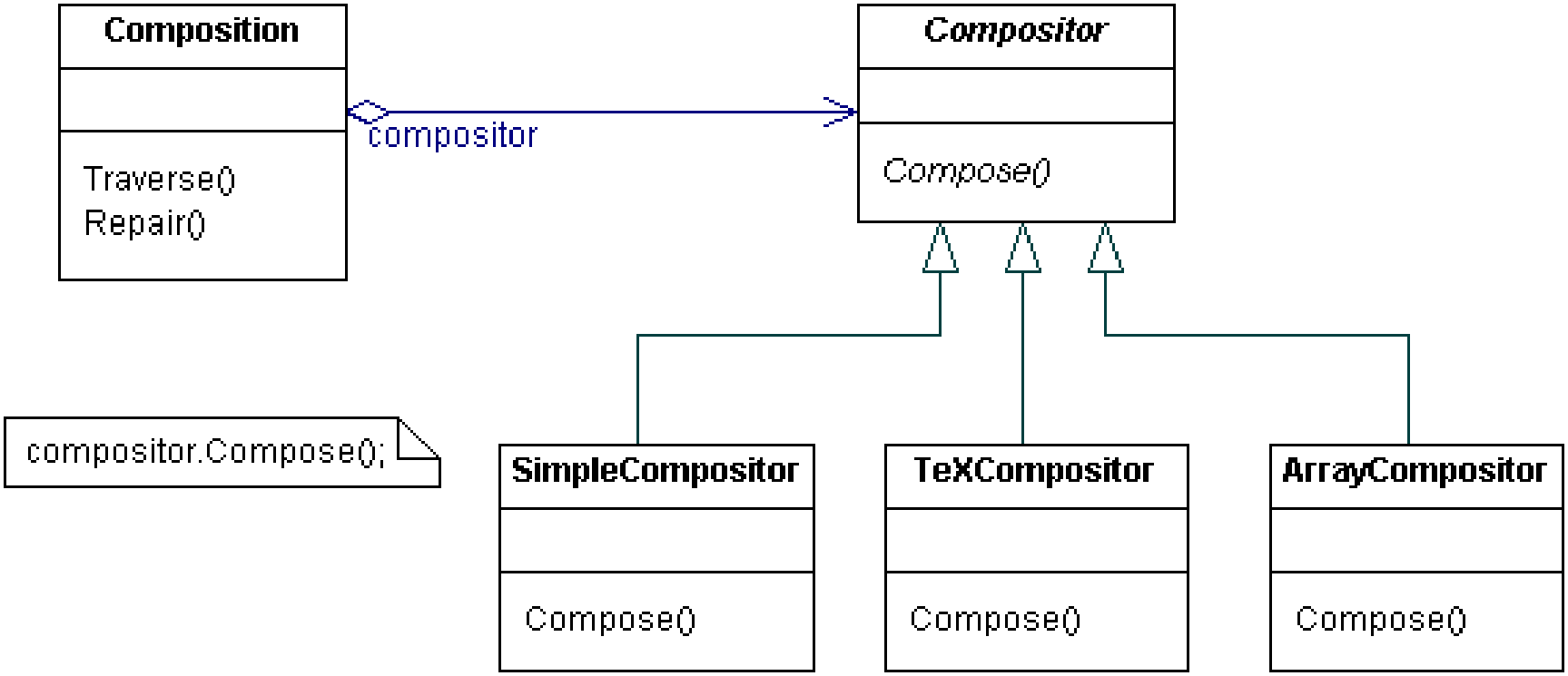
Strategy

- Propósito
  - Definir uma família de algoritmos, encapsular cada um, e fazê-los intercambiáveis.
  - Strategy permite que algoritmos variem independentemente entre clientes que os utilizam.

- Motivação: existem muitos algoritmos para quebrar um texto em linhas.
  - Separar o algoritmo de quebra de linha do cliente simplifica a codificação do mesmo.
  - Diferentes algoritmos são apropriados para diferentes aplicações.
  - Tornar o algoritmo parte do cliente dificulta adição de novos algoritmos.

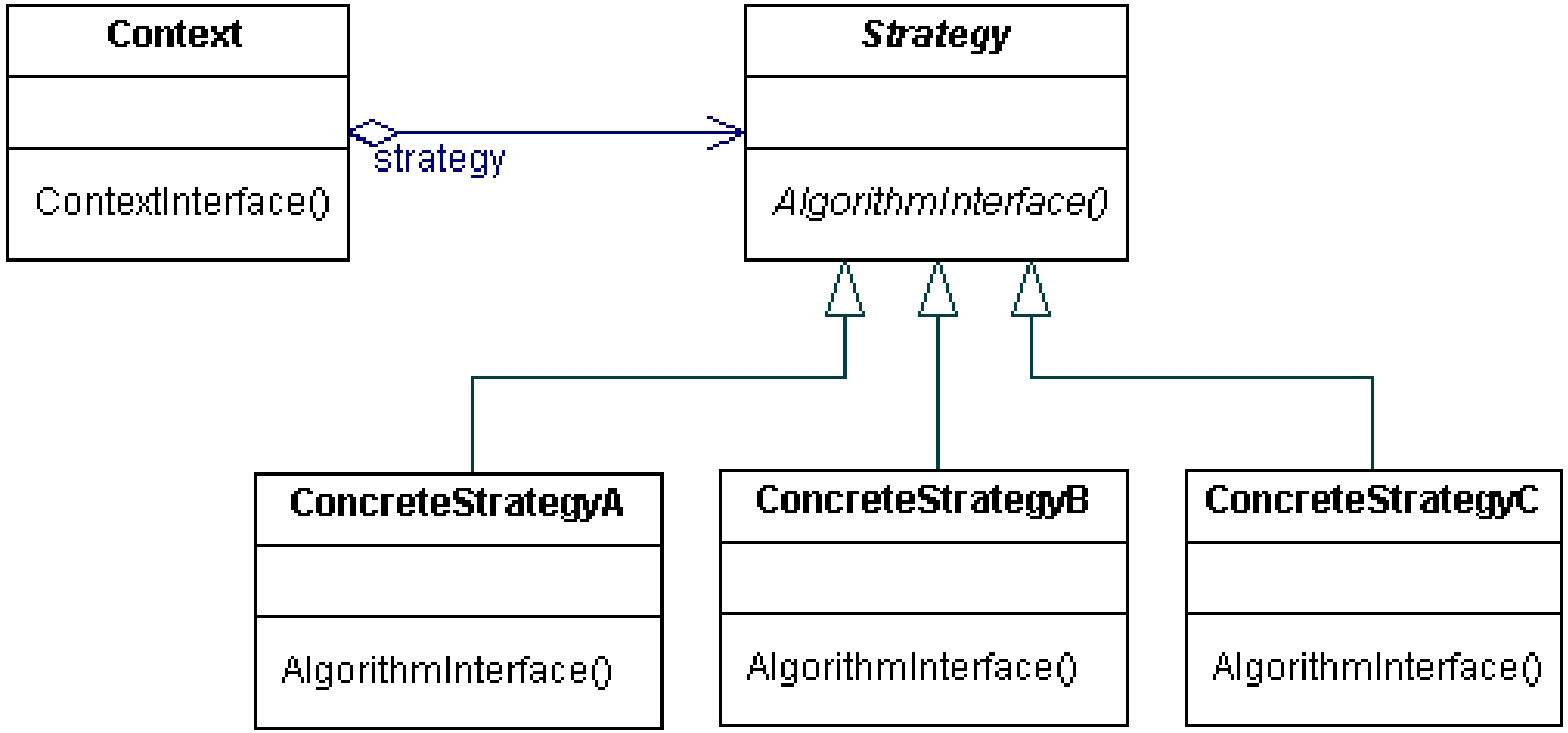
Strategy

- Motivação



Strategy

- Estrutura



Strategy

- Participantes:
  - Strategy (Compositor)
    - Define uma interface comum para todos os algoritmos suportados. Context usa esta interface para chamar o algoritmo definido por uma ConcreteStrategy.
  - ConcreteStrategy (SimpleCompositor, TeXCompositor)
    - Implementa o algoritmo usando a interface de Strategy.
  - Context (Composition)
    - É configurado com um objeto ConcreteStrategy;
    - Mantém uma referência para um objeto Strategy;
    - Pode definir uma interface que permite a Strategy acessar seus dados.

Strategy

```
public class Guerra {
   Estrategia acao;
   public void definirEstrategia() {
     if (inimigo.exercito() > 10000) {
        acao = new AliancaVizinho();
     } else if (inimigo.isNuclear()) {
        acao = new Diplomacia();
     } else if (inimigo.hasNoChance()) {
        acao = new AtacarSozinho();
     }
   }
   public void declararGuerra() {
      acao.atacar();
   }
   public void encerrarGuerra() {
      acao.concluir();
   }
}
```

```
public interface Estrategia {
   public void atacar();
   public void concluir();
}
```

```
public class AtacarSozinho
         implements Estrategia {
   public void atacar() {
      plantarEvidenciasFalsas();
      soltarBombas();
      derrubarGoverno();
   }
   public void concluir() {
      estabelecerGovernoAmigo();
   }
}
```

```
public class AliancaVizinho
         implements Estrategia {
   public void atacar() {
      vizinhoPeloNorte();
      atacarPeloSul();
      ...
   }
   public void concluir() {
      dividirBeneficios(...);
      dividirReconstrução(...);
   }
}
```

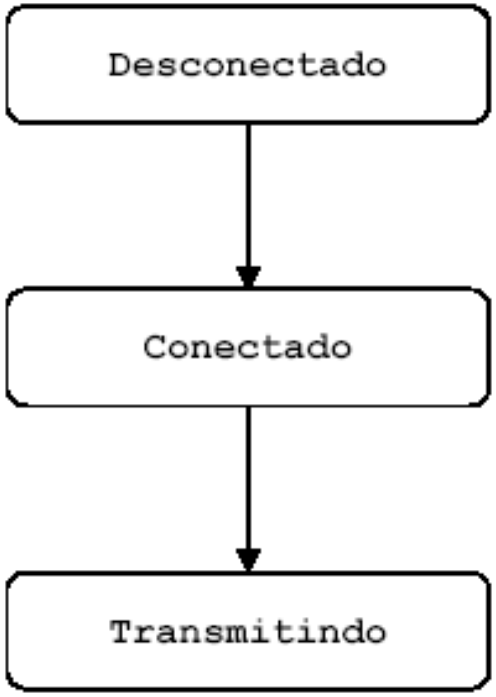
```
public class Diplomacia
         implements Estrategia {
   public void atacar() {
      recuarTropas();
      proporCooperacaoEconomica();
      ...
   }
   public void concluir() {
      desarmarInimigo();
   }
}
```

State

- Propósito
  - Permitir a um objeto alterar o seu comportamento quanto o seu estado interno mudar.
  - O objeto irá aparentar mudar de classe.

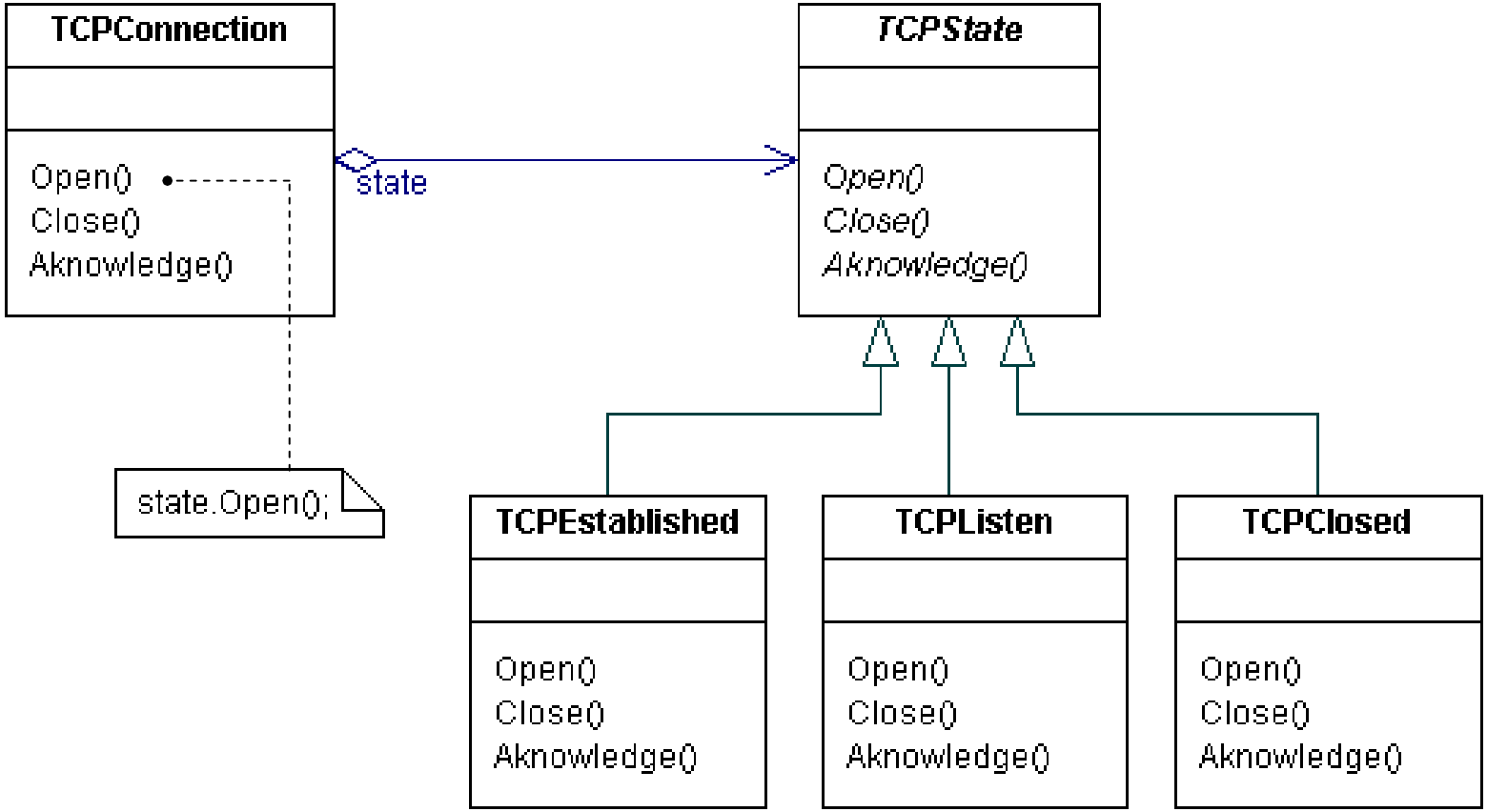
State

- Motivação: usar objetos para representar estados e polimorfismo para tornar transparente a execução de tarefas dependentes de estado.



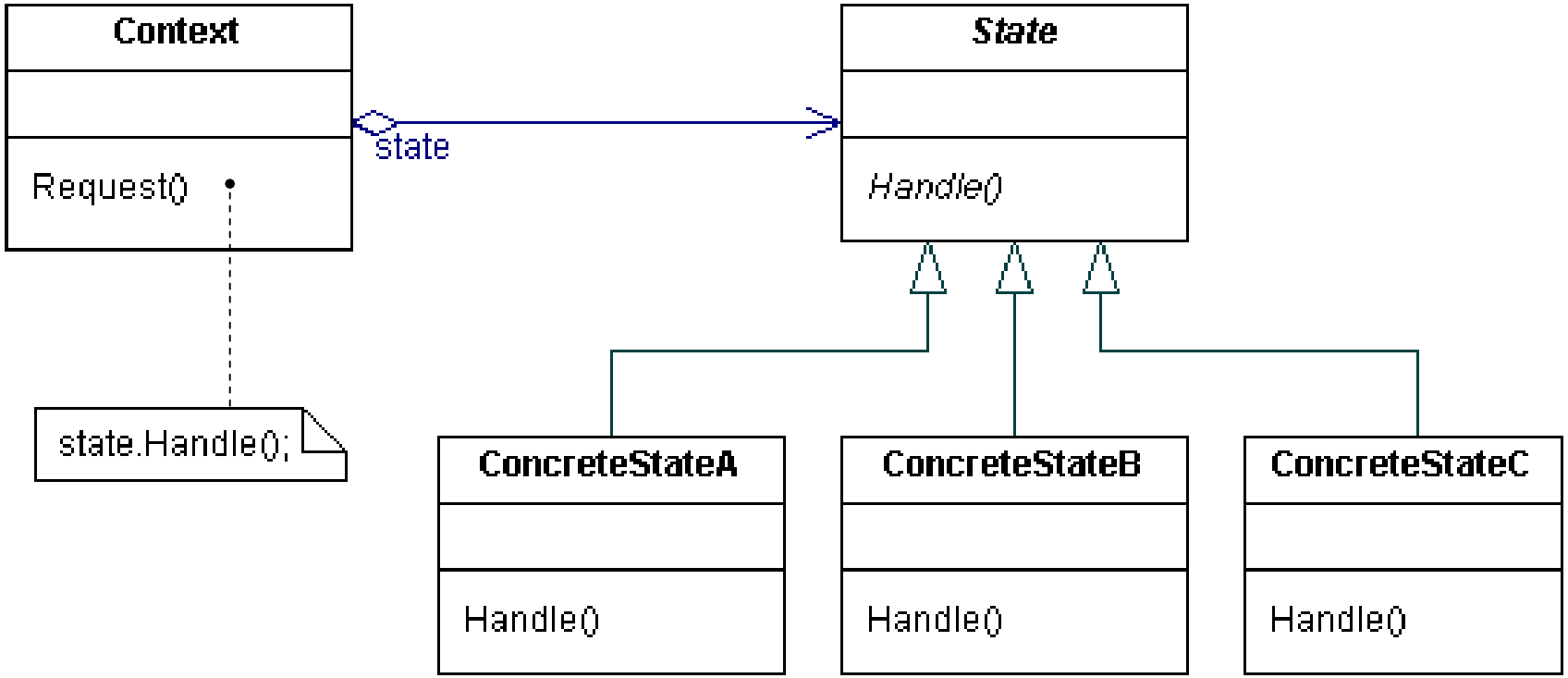
State

- Motivação



State

- Estrutura

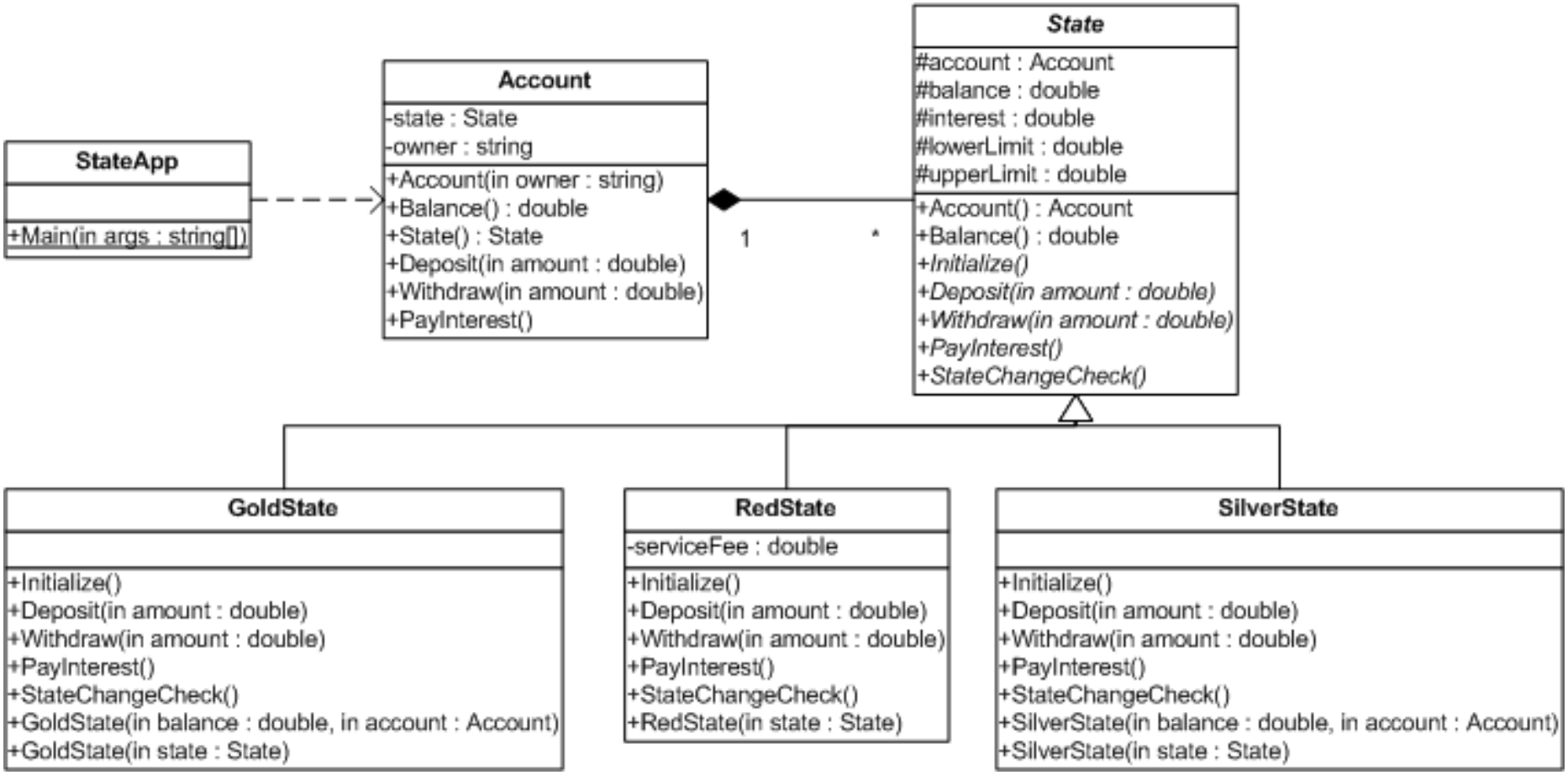


State

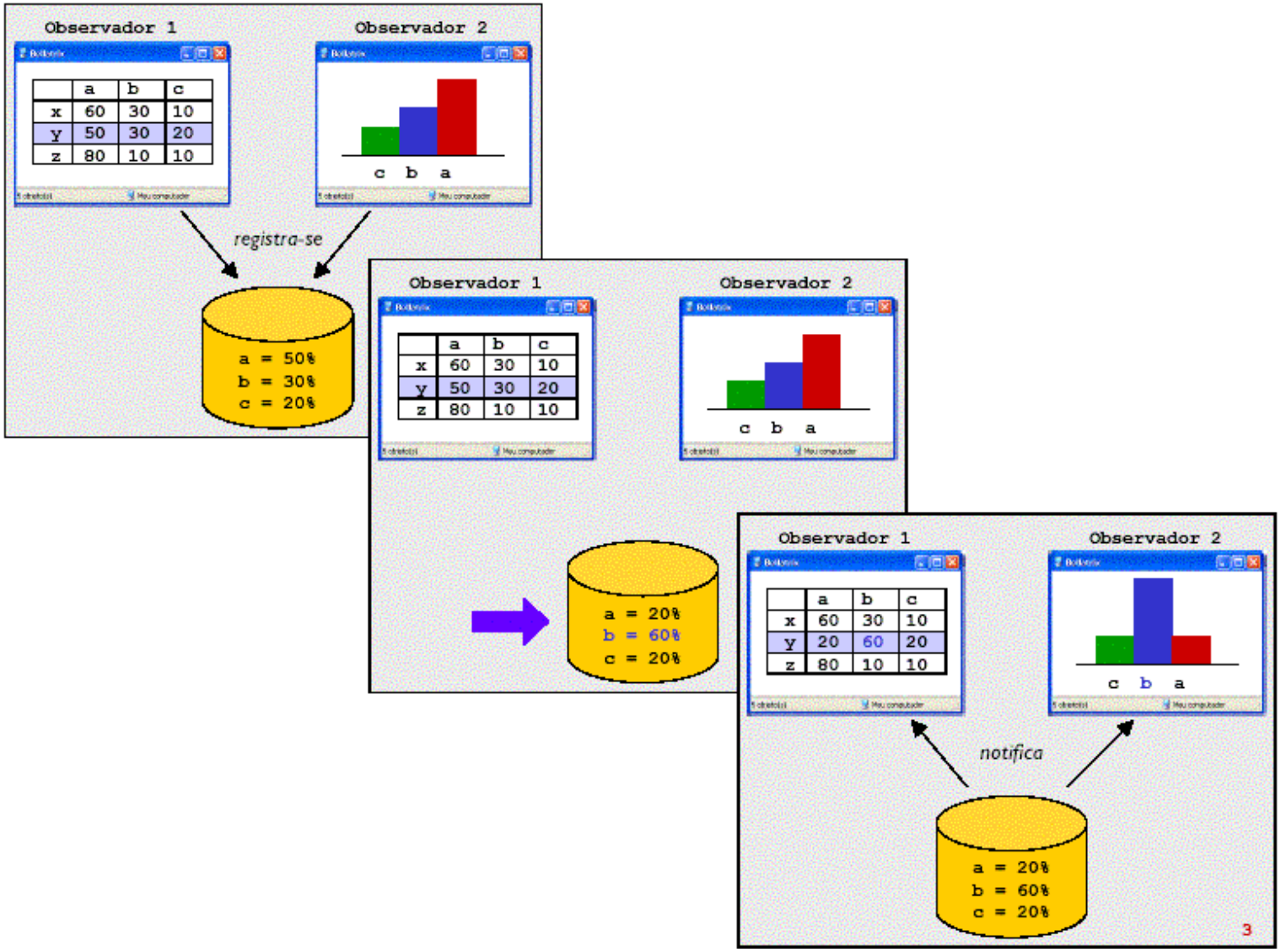
- Participantes:
- Context
  - Define a interface de interesse para os clientes.
  - Mantém uma instância de uma subclasse ConcreteState que define o estado corrente.
- State
  - define uma interface para encapsular o comportamento associado com um estado particular do contexto
- ConcreteState
  - Implementa um comportamento associado a um estado do Context.

State

- Exemplo

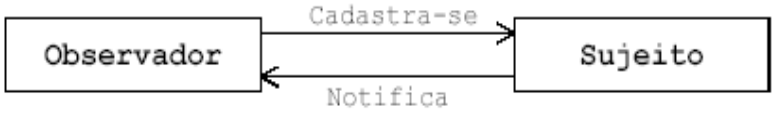


Observer


Observador 1
Observador 2
registra-se
a = 50%
b = 30%
c = 20%
Observador 1
Observador 2
a = 20%
b = 60%
c = 20%
Observador 1
Observador 2
notifica
a = 20%
b = 60%
c = 20%
3

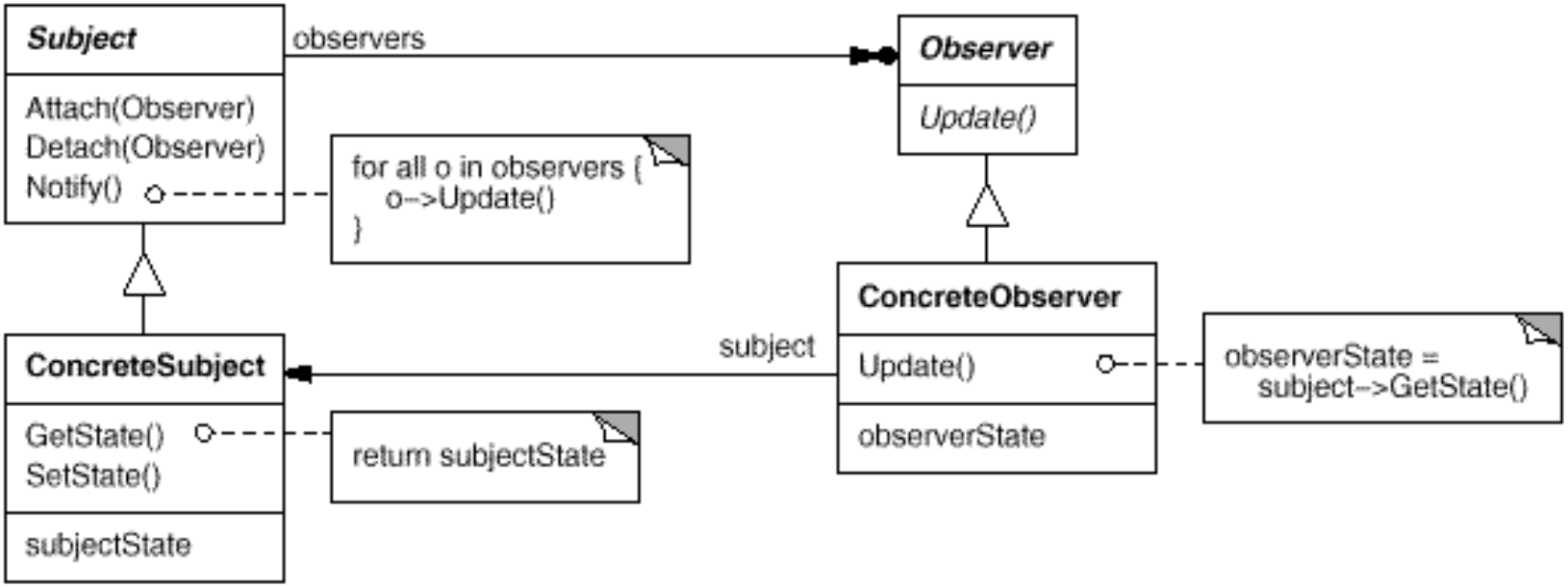
Observer

- Propósito
  - Dependência de um-para-muitos entre objetos
    - Quando um objeto muda de estado, todos seus dependentes são notificados e atualizados automaticamente
- Aplicabilidade
  - Abstração tem dois aspectos, um dependente do outro
    - Encapsular estes aspectos em objetos separados permite variação e reuso independentemente
  - Mudança em um objeto requer alterar outros
    - Não se sabe quantos objetos precisam ser alterados
  - Objeto capaz de notificar outros objetos sem presumir quem são esses objetos



Observer

- Estrutura



Observer

- Participantes
  - Subject
    - Conhece seu Observer
    - Qualquer número de objetos Observer podem observar um Subject
    - Provê uma interface para acoplar e desacoplar objetos Observer
  - ConcreteSubject
    - Guarda o estado de interesse para ConcreteObserver
    - Envia uma notificação para seu Observer quando seu estado muda
  - Observer
    - Define uma interface de atualização para objetos que devem ser notificados sobre mudanças em um Subject
  - ConcreteObserver
    - Mantém uma referência para um objeto ConcreteSubject
    - Guarda o estado que deve ficar consistente com o de Subject
    - Implementa o Observer atualizando a interface para manter seu estado consistente com o de Subject

Observer

- Conseqüências
  - Acoplamento abstrato entre Sujeito e Observador
  - Suporte a comunicação em broadcast (mensagem que todos os observadores enxergam).
  - Atualizações inesperadas

Observer

```
public class ConcreteObserver
             implements Observer {

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

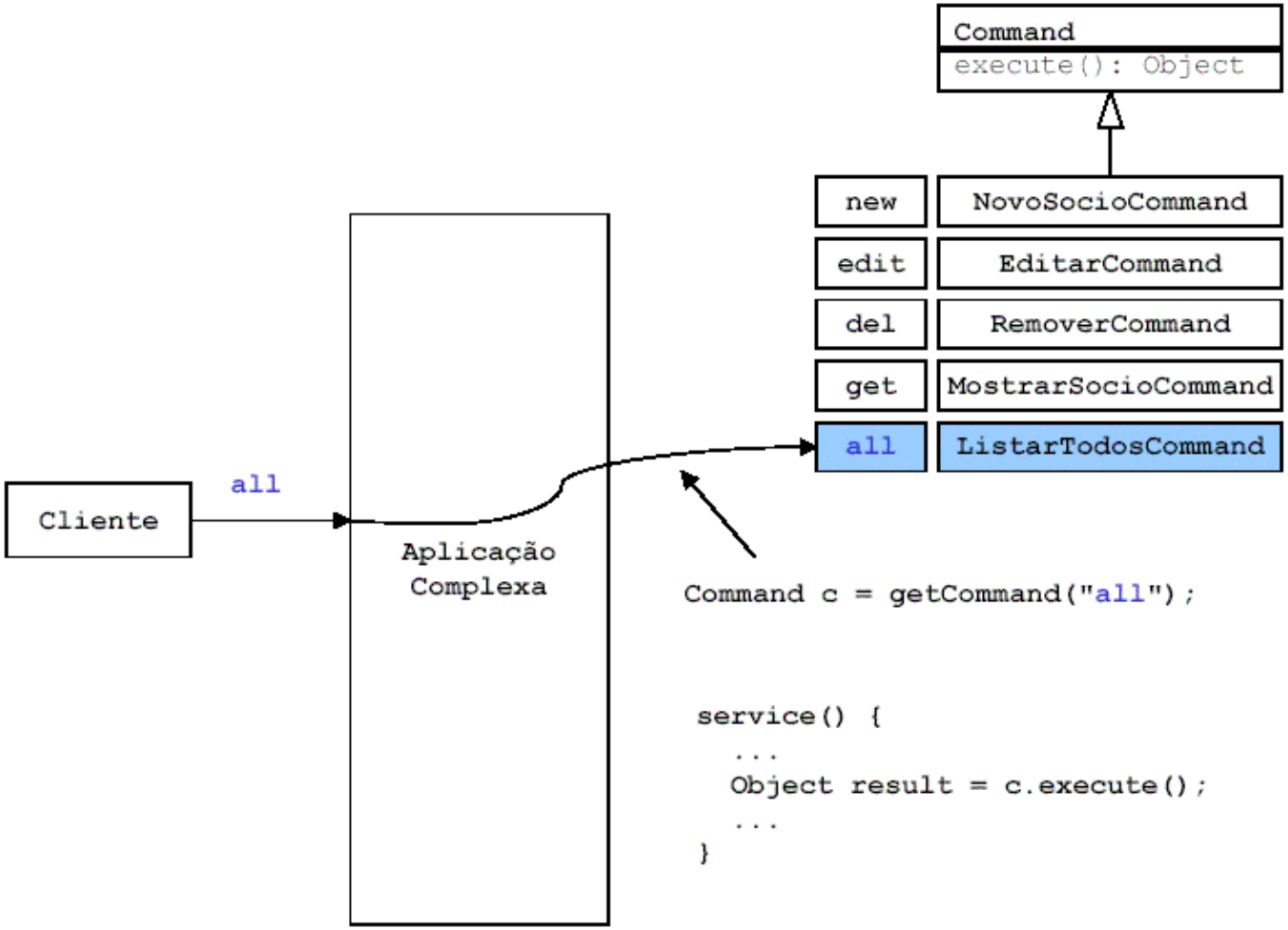
```
public class ObservableData
             extends Observable {
  private Object myData;

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

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

```
public class Observable {
  Observer 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);
}
```

Command

- Motivação

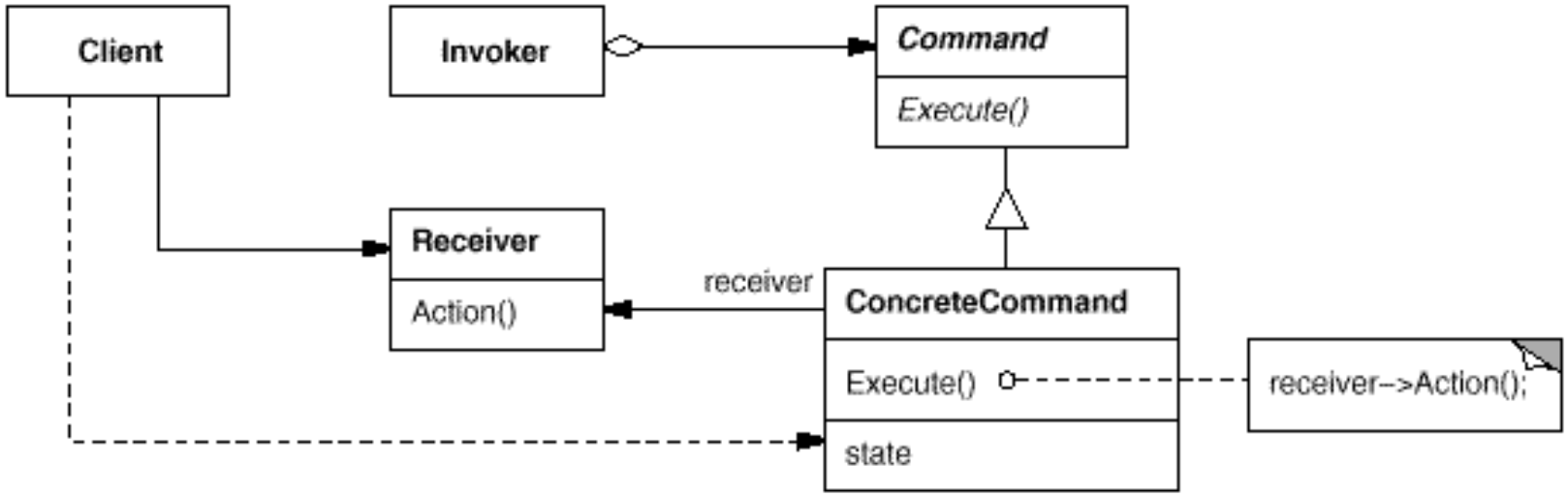


Command

- Propósito
  - Encapsular requisição como objeto
    - Permite parametrizar clientes com diferentes requisições
  - Dar suporte a operações que não podem ser desfeitas
- Aplicabilidade
  - Parametrizar objetos por ação a ser realizada
  - Especificar, enfileirar e executar requisições em diferentes momentos
  - Suportar "desfazer"
  - Suportar log de alterações
    - Podem ser reaplicadas caso o sistema falhe
  - Estruturar o sistema em operações de alto nível construídas sobre operações primitivas

Command

- Estrutura



Command

- Participantes
  - Command
    - Define interface para a execução de uma operação
  - ConcreteCommand
    - Define uma vinculação entre um objeto Receiver e uma ação
    - Implementa Execute através da invocação da(s) correspondente(s) operação(ões) no Receiver
  - Client
    - Cria um objeto ConcreteCommand e estabelece o seu receptor (Receiver)
  - Invoker
    - Solicita ao Command a execução da solicitação
  - Receiver
    - Sabe como executar as operações associadas a uma solicitação
      - Qualquer classe pode funcionar como um receiver

Command

- Conseqüências
  - Desacopla objeto que invoca operação do que sabe realizá-la
  - Comandos são objetos de "primeira classe"
  - Comandos podem ser reunidos para fazer um comando composto
  - Facilidade de adicionar novos comandos

Command

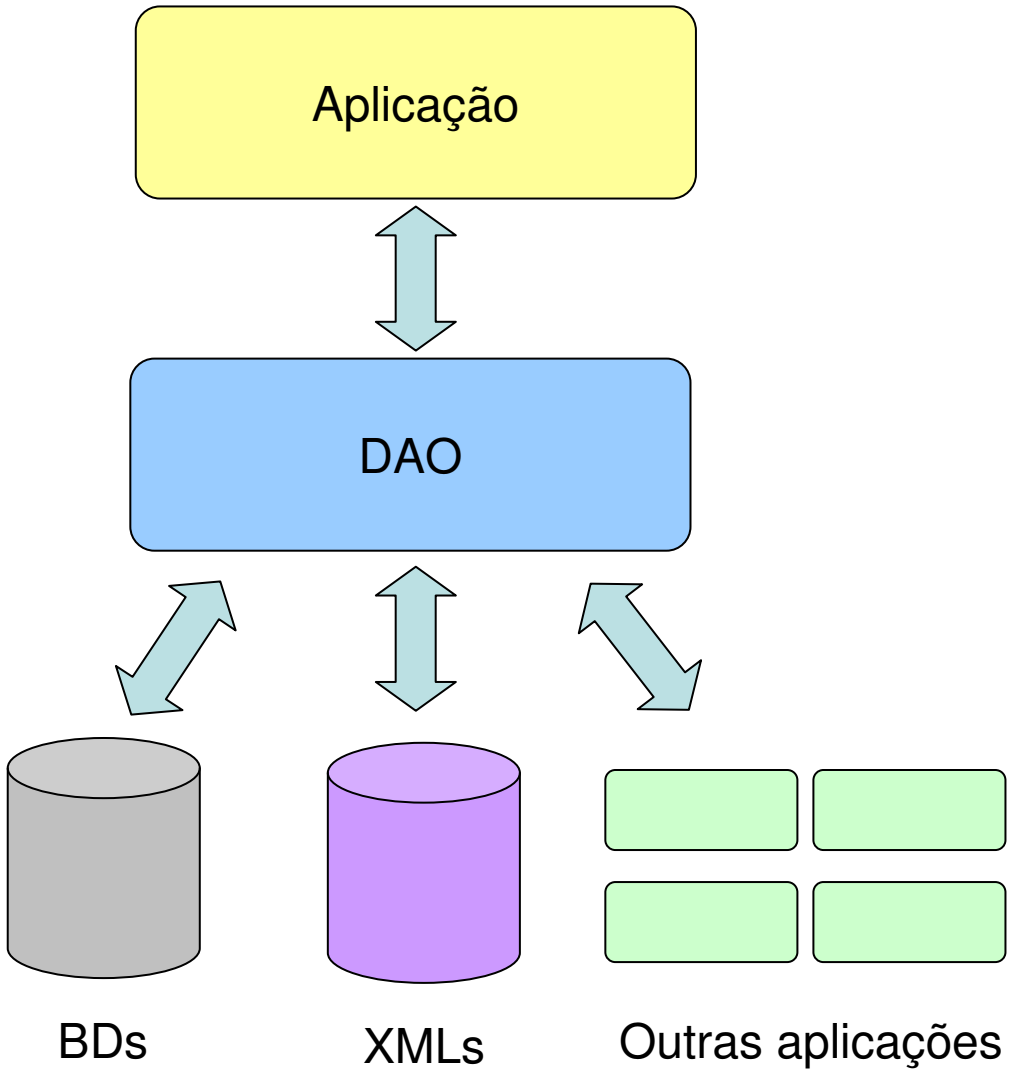
```
public interface Command {
   public Object execute(Object arg);
}
```

```
public class Server {
  private Database db = ...;
  private HashMap cmds = new HashMap();

  public Server() {
    initCommands();
  }

  private void initCommands() {
    cmds.put("new", new NewCommand(db));
    cmds.put("del",
                  new DeleteCommand(db));
    ...
  }

  public void service(String cmd,
                      Object data) {
    ...
    Command c = (Command)cmds.get(cmd);
    ...
    Object result = c.execute(data);
    ...
  }
}
```

```
public class NewCommand implements Command {

  public NewCommand(Database db) {
    this.db = db;
  }

  public Object execute(Object arg) {
    Data d = (Data)arg;
    int id = d.getArg(0);
    String nome = d.getArg(1);
    db.insert(new Member(id, nome));
  }
}
```

```
public class DeleteCommand implements Command {

  public DeleteCommand(Database db) {
    this.db = db;
  }

  public Object execute(Object arg) {
    Data d = (Data)arg;
    int id = d.getArg(0);
    db.delete(id);
  }
}
```

Data Access Object - DAO



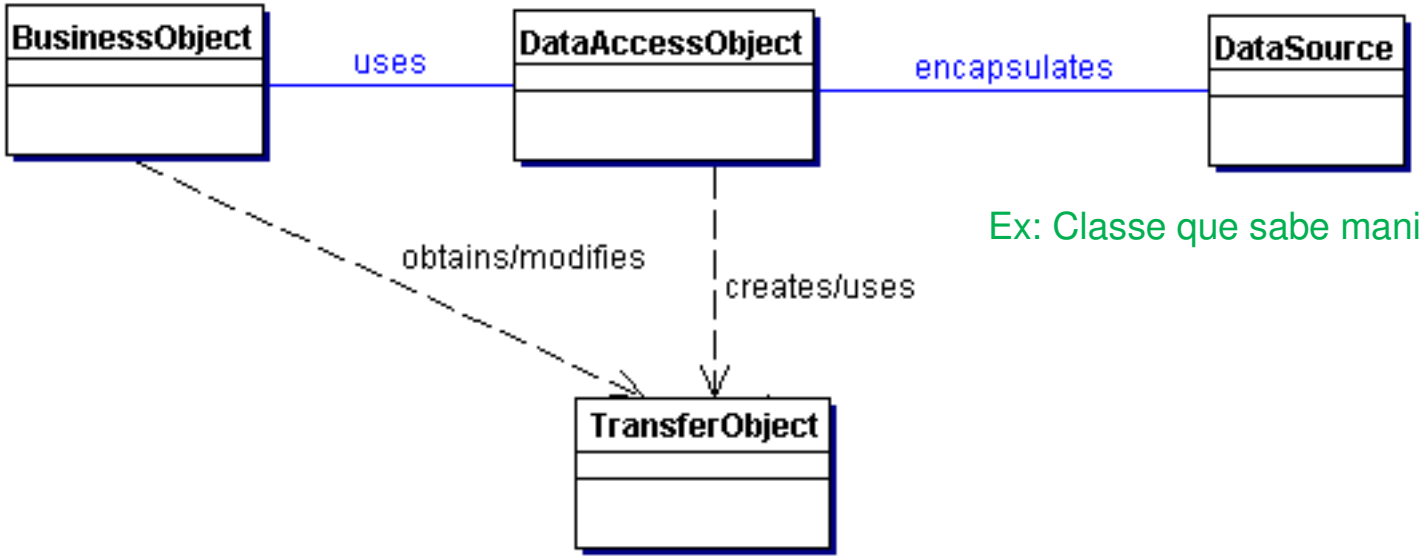
DAO

- Propósito
  - Mediador entre as aplicações e a base de dados
  - Tradutor dos mundos

- Aplicabilidade
  - Base de dados fornece dados para alguma aplicação
    - DAO converte os dados para serem manipulados
  - Aplicação fornece dados para a base de dados
    - DAO realiza a tradução para armazenamento, por exemplo.

DAO

- Estrutura



Ex: Classe que sabe manipular cliente no BD

Ex: classe cliente da aplicação

DAO

- Participantes
  - BusinessObject
    - Requisita acesso para armazenar ou requisitar algum dado de algum base.
  - DataAccessObject (DAO)
    - Oferece serviços para o BusinessObject de forma transparente.
  - DataSource
    - Representa a fonte de dados (ex: BD, outro sistema, repositório XML, etc). Acessada pelo DAO.
  - TransferObject
    - Usado para representar os dados obtidos pelo DAO e para serem compreendidos pelo BusinessObject.

DAO

- Conseqüências
  - Organização na forma de prover e requisitar informações localizadas em bases de dados.
  - Simplificação na manutenção
  - Baixo acoplamento.

DAO

Cliente da aplicação

```
public class BusinessObject
{
        void execute()
        {
                Dao dao = new Dao();

                Data data = new Data();
                data.setName("João");
                ...
                dao.insert(data);

                Data data = dao.getClient("123");
                ...
        }
}
```

Classe que sabe manipular cliente

```
public class DataSource
{
        private String client;
        ...
        public void setClient(String name)
        {
                this.client = name;
        }
        ...
        public String getClient()
        {
                return client;
        }
        ...
}
```

```
public class DAO
{
        public void insert(Data data)
        {
                DataSource ds = new DataSource();
                ds.setClient(data.getName());
                ds.setEndereco(data.getRua() + data.getNum() + data.getComplemento());
                ...
        }

        public Data getClient(String id)
        {
                ...
                Data data = new Data();
                data.setName(ds.getClient());
                ...
        }
}
```