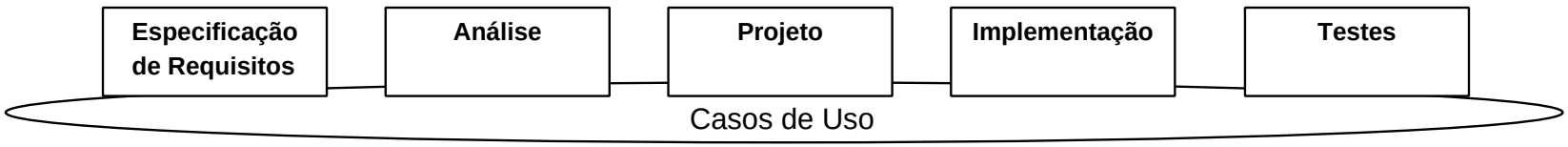
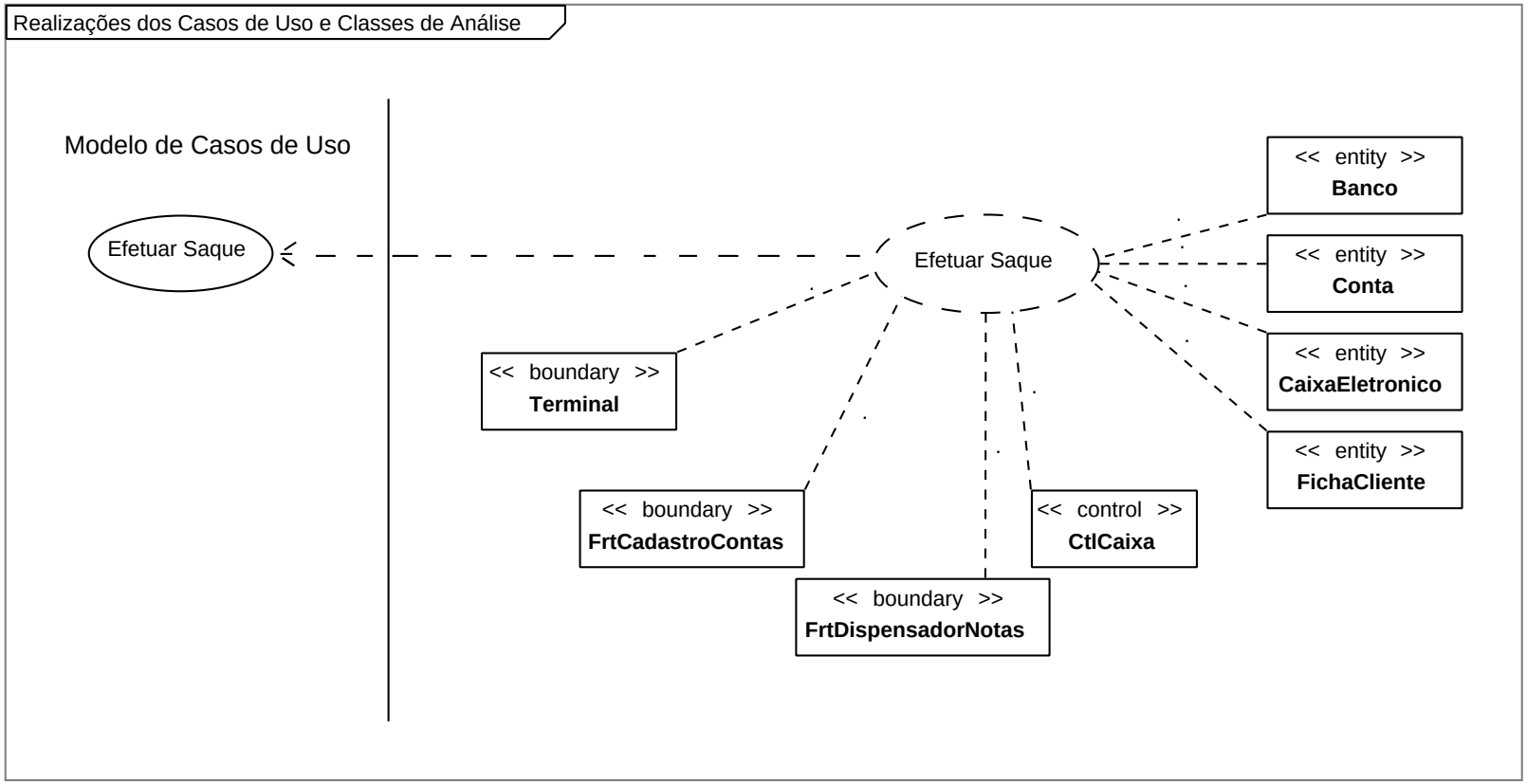
Análise Orientada a Objetos

- A análise OO foca no entendimento dos requisitos, conceitos e operações relacionados com o sistema.
- Enfatiza **o que** deve ser feito.
- Resultados produzidos:
  - um diagrama de classes de análise, que descreve os conceitos relevantes do sistema; e
  - um conjunto de diagramas de interação, que representam os processos aos quais o sistema deve dar suporte.

Papel Unificador dos Casos de Uso



Realizações dos Casos de Uso e Classes de Análise



Projeto Orientado a Objetos (I)

- O foco do desenvolvimento muda do domínio do problema para o domínio da solução.
- Enfatiza a maneira **como** os requisitos serão satisfeitos pelo sistema.
- Resultados produzidos:
  - um diagrama de classes de projeto, similar ao diagrama de classes de análise, só que mais detalhado e focado no domínio da solução; e
  - um conjunto de realizações de casos de uso, que definem a maneira como as classes de projeto colaboram a fim de prover o comportamento especificado pelos casos de uso.

Projeto Orientado a Objetos (II)

- O projeto OO costuma ser dividido em duas etapas: **projeto arquitetural** e **projeto detalhado**.
- O projeto arquitetural define a estrutura geral do sistema, sua quebra em componentes, os relacionamentos entre esses componentes e a maneira como interagem.
- O projeto detalhado descreve as classes do sistema, seus relacionamentos e as interações entre suas instâncias em tempo de execução.
- Focaremos nossa atenção no projeto arquitetural.

Arquitetura de Software

- Reflete o conjunto de decisões que devem ser tomadas primeiro no projeto desse sistema; aquelas que são mais difíceis de mudar em etapas posteriores do desenvolvimento.
- Está relacionada com a complexidade global do sistema e com a integração dos seus subsistemas.
- A elaboração da arquitetura de software de um sistema envolve decisões relativas a: particionamento do sistema em subsistemas menores, determinação do fluxo de controle interno da arquitetura, definição de camadas e protocolos de interação entre as mesmas, alocação de software/hardware.

Propriedades Não-Funcionais de Arquiteturas de Software (I)

- Uma **propriedade funcional** lida com algum aspecto particular da funcionalidade do sistema e está usualmente relacionada a um requisito funcional especificado.
- Uma **propriedade não-funcional** (ou administrativa) denota uma característica de um sistema que não é coberta pela sua descrição funcional.

Propriedades Não-Funcionais de Arquiteturas de Software (I)

- Propriedades não-funcionais tipicamente estão relacionadas a aspectos como confiabilidade, adaptabilidade, interoperabilidade, usabilidade, persistência, manutenabilidade, distribuição e segurança.
- As decisões de projeto que devem ser tomadas no desenvolvimento de uma arquitetura de software geralmente envolvem vários aspectos relacionados com as suas propriedades não-funcionais

Arquitetura de Software

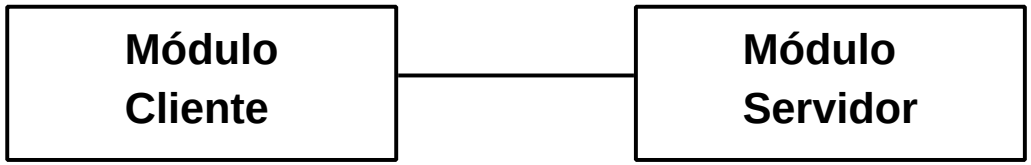
- Representa o sistema em termos dos seus módulos e das restrições de comunicação entre eles
- As estruturas básicas da arquitetura são:
  - **Componentes** - Entidades funcionais do sistema. Implementam os requisitos especificados.
  - **Conectores** - Entidades responsáveis pela ligação entre os componentes.
  - **Configuração** - Maneira como os componentes e conectores se integram em uma determinada disposição.

Estilo Arquitetural

- Caracteriza uma família de sistemas que são relacionadas pelo **compartilhamento de propriedades estruturais e comportamentais (semântica)**;
- Um estilo de arquitetura define um **vocabulário para os elementos da arquitetura**: componentes e conectores;
- Além disso, apresenta ainda **regras e restrições** sobre a combinação dos componentes arquiteturais;

Estilo Arquitetural Cliente-Servidor

- Possui dois módulos bem definidos:
  - Módulo **Cliente**
  - Módulo **Servidor**



Escalas de Arquiteturas de Software

- **Nível da aplicação**: preocupa-se diretamente com os requisitos funcionais da aplicação.
- **Nível do sistema**: cuida das conexões entre diversas aplicações. Este nível fornece uma infraestrutura de apoio para o nível de aplicação.
- **Nível de empresa**: compreende vários sistemas, onde cada sistema pode englobar diversas aplicações.
- **Nível global**: é a maior das escalas de arquitetura, podendo compreender diversas empresas. É responsável pelas questões de impacto de software que atravessam as fronteiras entre empresas.

As Visões da Arquitetura de Software

- Surgiram em resposta à insistência de alguns autores em representar toda a arquitetura do sistema através de uma única representação.
- Uma arquitetura de software deve ser representada através de quatro visões diferentes e ortogonais entre si: (i) lógica, (ii) de componentes, (iii) de processos e (iv) de implantação
- A visão de casos de uso faz a integração entre as outras quatro.
- Esse modelo para a representação de arquiteturas é conhecido como $\mathbf{Modelo\ 4{+}1}$.

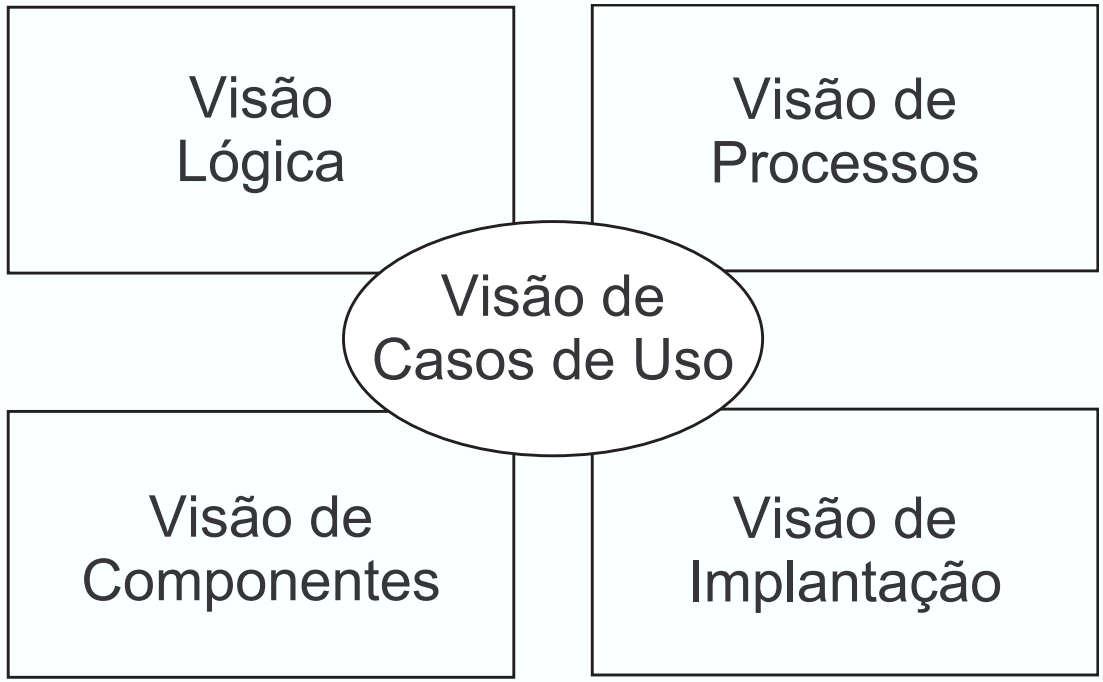
O Modelo 4+1 para a Representação de Arquiteturas (I)

- **Visão Lógica**: decompõe o sistema em um conjunto de abstrações, na forma de objetos ou classes que se relacionam.
- **Visão de Componentes ou Desenvolvimento/Implementação**: foca na organização do sistema em módulos de código
- **Visão de Processos ou Concorrência**: lida com a divisão do sistema em processos e processadores e trata de aspectos relacionados à execução concorrente do sistema.

O Modelo 4+1 para a Representação de Arquiteturas (II)

- **Visão de Implantação ou Física**: representa o sistema como um conjunto de elementos que se comunicam e que são capazes de realizar processamento (como computadores e outros dispositios).
- **Visão de Casos de Uso ou Cenários**: é responsável por integrar as outras quatro e descreve a funcionalidade provida pelo sistema aos seus atores.

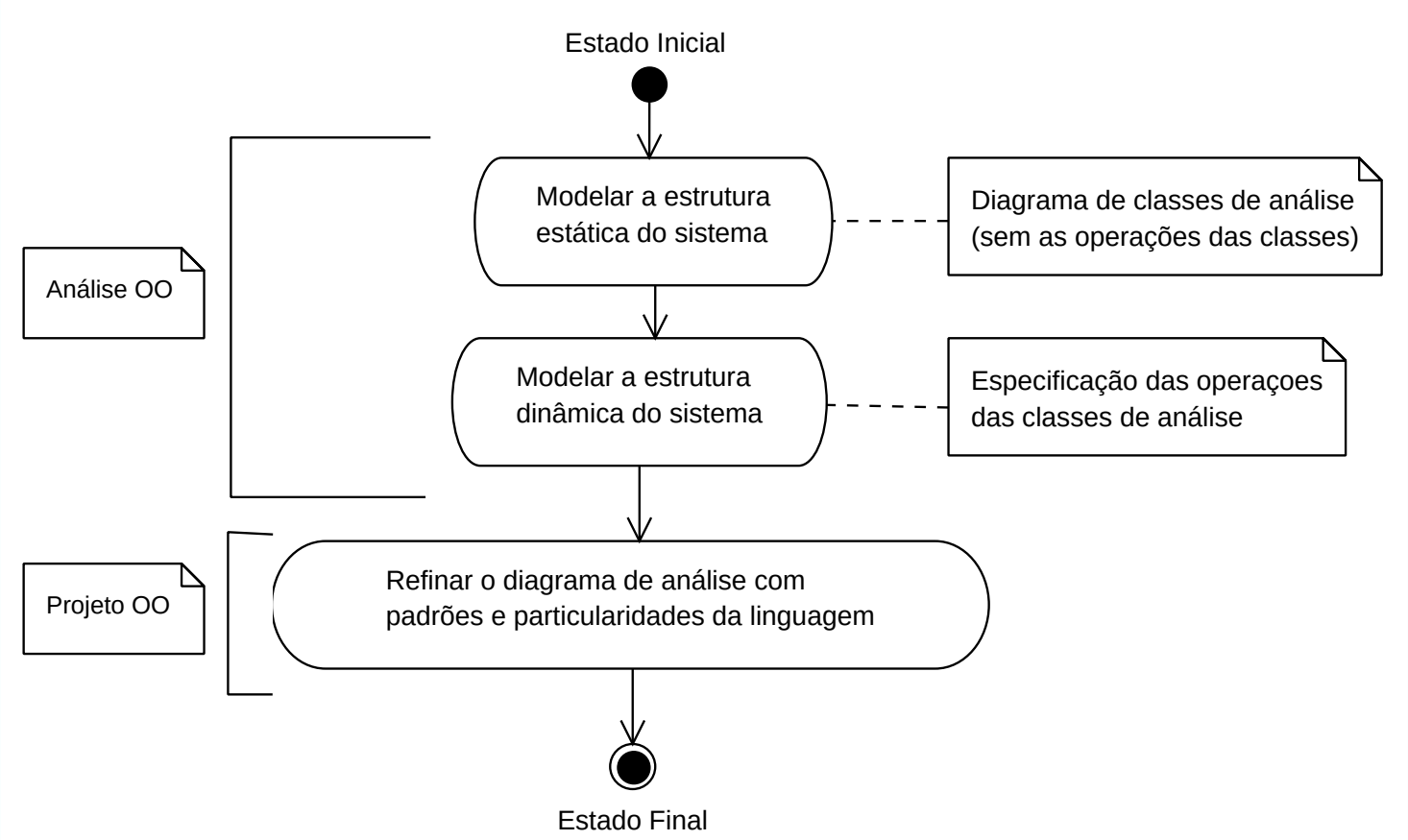
O Modelo 4+1 para a Representação de Arquiteturas (III)



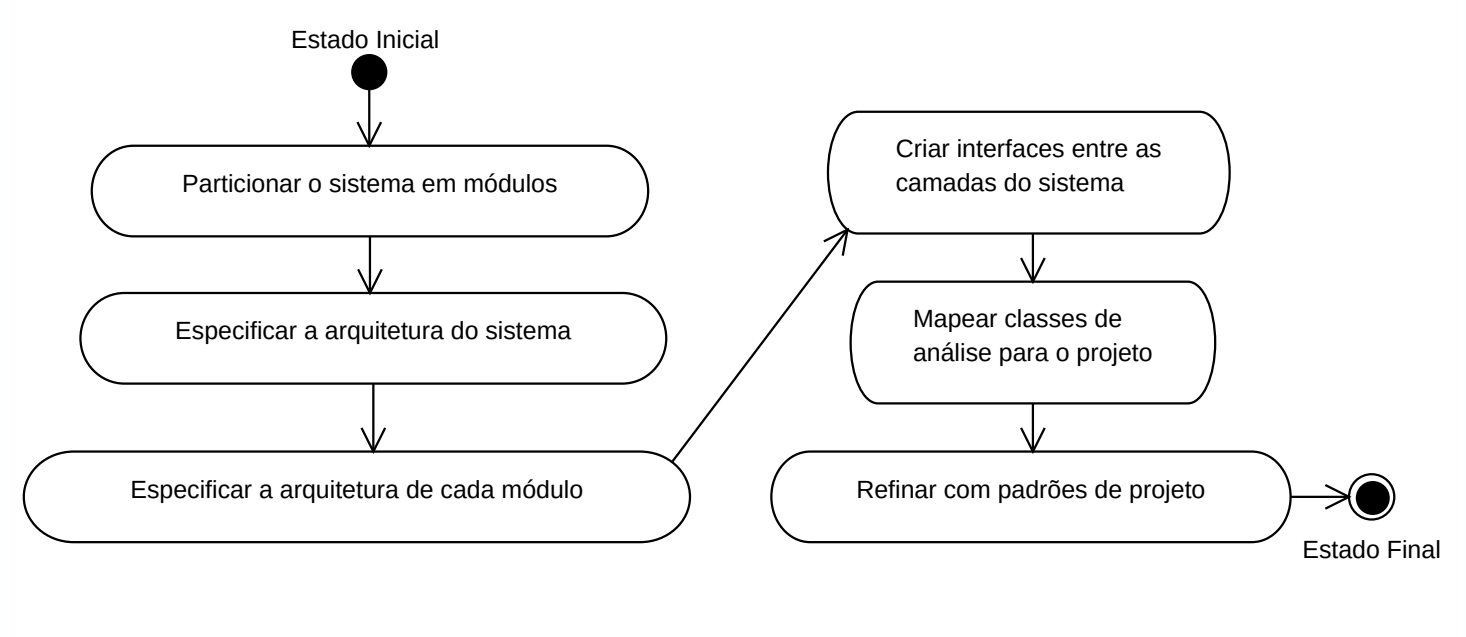
Resumo das Visões do Modelo 4+1

Visão	**Requisito Principal**	**Participante Principal**
Visão Lógica	Grupos funcionais	Usuário Final
Visão de Processo	Desempenho Escalabilidade Transferência de dados (taxa)	Integrador do Sistema
Visão de Desenvolvimento	Topologia do sistema Comunicação	Engenheiro de Sistema
Visão Física	Gerenciamento do Software Distribuição e instalação	Programador
Visão de Casos de Uso	Comportamento (funcionalidades)	Analista de sistemas Analista de testes

Atividades da Passagem da Análise ao Projeto (I)



Atividades da Passagem da Análise ao Projeto (II)



1: Particionar o Sistema em Módulos

- O modelo de análise deve ser particionado em conjuntos, **coesos** e **fracamente acoplados** de classes.
- Esses conjuntos são chamados subsistemas.
- Os componentes de um subsistema Eles podem estar envolvidos na realização de um mesmo conjunto de requisitos funcionais, residir dentro de um mesmo produto de hardware ou gerenciar um mesmo tipo de recurso.
- Subsistemas são identificados pelos serviços definidos por suas interfaces.
- Um subsistema provê serviços e utiliza serviços providos por outros subsistemas.

Critérios para Particionar um Sistema

- O subsistema deve ter uma interface bem definida através da qual ocorre toda a comunicação com o resto do sistema.
- Com exceção de um pequeno número de "classes de comunicação", as classes em um subsistema devem colaborar apenas com outras classes dentro do próprio subsistema.
- O número de subsistemas deve ser pequeno.
- Um subsistema pode ser particionado internamente a fim de reduzir sua complexidade.

2: Especificar a Arquitetura do Sistema

- Três etapas:
  1. Identificar os módulos funcionais que constituem o sistema (componentes);
  2. Estabelecer regras de interações entre esses componentes (conectores);
  3. Especificar as associações entre os componentes do sistema (configuração);
- Para isso, pode-se consultar uma lista de padrões.

Padrões (I)

- Soluções para problemas comuns encontrados durante o desenvolvimento.
- Validadas através do uso em diversas circunstâncias distintas.
- Especificadas de uma maneira estruturada que descreve tanto a solução quanto o problema que ela se propõe a resolver.
- Associadas a um nome.

Padrões (II)

- Segundo Christopher Alexander:

  "cada padrão descreve um problema que ocorre recorrentemente em nosso ambiente e então descreve uma solução para o problema, de tal maneira que você pode usar essa solução um milhão de vezes, sem nunca utilizá-la duas vezes da mesma maneira".

Padrões (III)

- Padrões são encontrados em diversos "sabores" diferentes:
  - Padrões para atribuição de responsabilidades;
  - Padrões de análise.
  - Padrões arquiteturais;
  - Padrões de projeto;
  - *Idioms*, ou padrões de implementação;

Estrutura de um Padrão

1. Nome;
2. Problema;
3. Solução;
4. Consequências;
5. Exemplos (opcional).

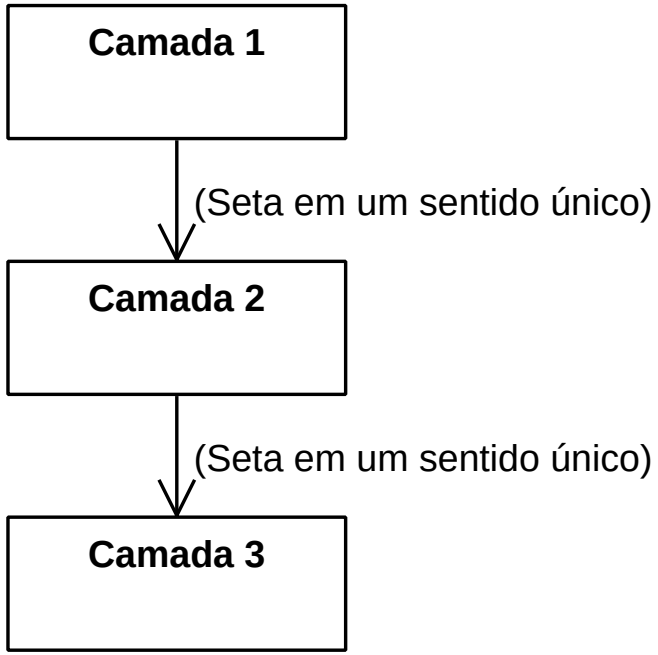
A Escolha de Um Padrão Arquitetural

- Um padrão arquitetural relaciona um estilo arquitetural a um grupo de sistemas (ou requisitos funcionais) que ele é indicado;
- A especificação da arquitetura pode consistir na escolha de um padrão (ou estilo) arquitetural adequado ao sistema;
- Seguir um determinado padrão arquitetural significa adotar estrutura conhecida e adequada para o sistema específico
- Sistemas de informação (sistemas comerciais em geral) possuem normalmente uma arquitetura em camadas, onde as camadas correspondem aos seus componentes

Divisão em Camadas (I)

- Um sistema também deve ser particionado em camadas.
- Essa divisão é ortogonal à divisão em subsistemas.
- Cada camada contém um ou mais subsistemas e é responsável por um conjunto de funcionalidades relacionadas.
- Subsistemas localizados na camada n usam os serviços providos pelos subsistemas da camada $n+1$ (abaixo) e provêm serviços para os subsistemas da camada $n-1$ (acima).
- As camadas $n-1$ e $n+1$ não "enxergam" uma à outra diretamente

Divisão em Camadas (II)



Exemplo: Arquitetura em três camadas

Divisão em Camadas (III)

- Uma camada esconde detalhes das implementações das camadas inferiores das camadas superiores.
- Uma estruturação comum para a construção de sistemas de informação consiste em usar três camadas, com as seguintes responsabilidades:

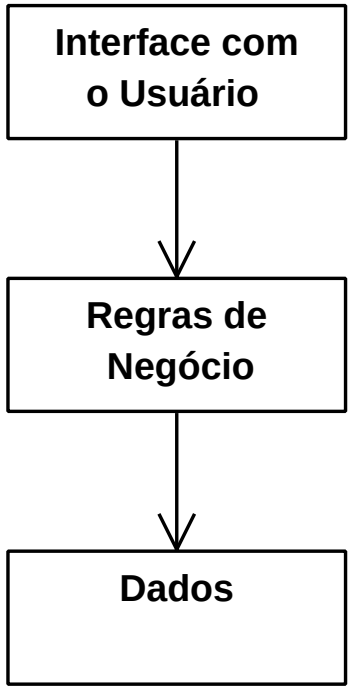
  (i) gerenciar a interação entre o sistema e o usuário;

  (ii) implementar as regras de negócio da aplicação;

  (iii) armazenar os dados da aplicação.

- Uma arquitetura de software com essas divisões é um exemplo conhecido de uma **Arquiteturas de Três Camadas**

Divisão em Camadas (IV)



Interfaces entre as Camadas

- Após particionar as classes de análise entre subsistemas e definir as camadas, o arquiteto deve definir as interfaces entre as camadas.
- O foco está na maneira como a comunicação entre os subsistemas em camadas adjacentes ocorrerá.
  - Quais classes representarão os pontos de acesso às camadas?
  - Quais tecnologias serão usadas para mediar a comunicação entre elas?
- Se a interface entre duas camadas é muito complexa e encapsula requisitos não-funcionais (por exemplo, distribuição), pode valer a pena criar uma nova camada.

Particionamento do Sistema de Caixa Automático (I)

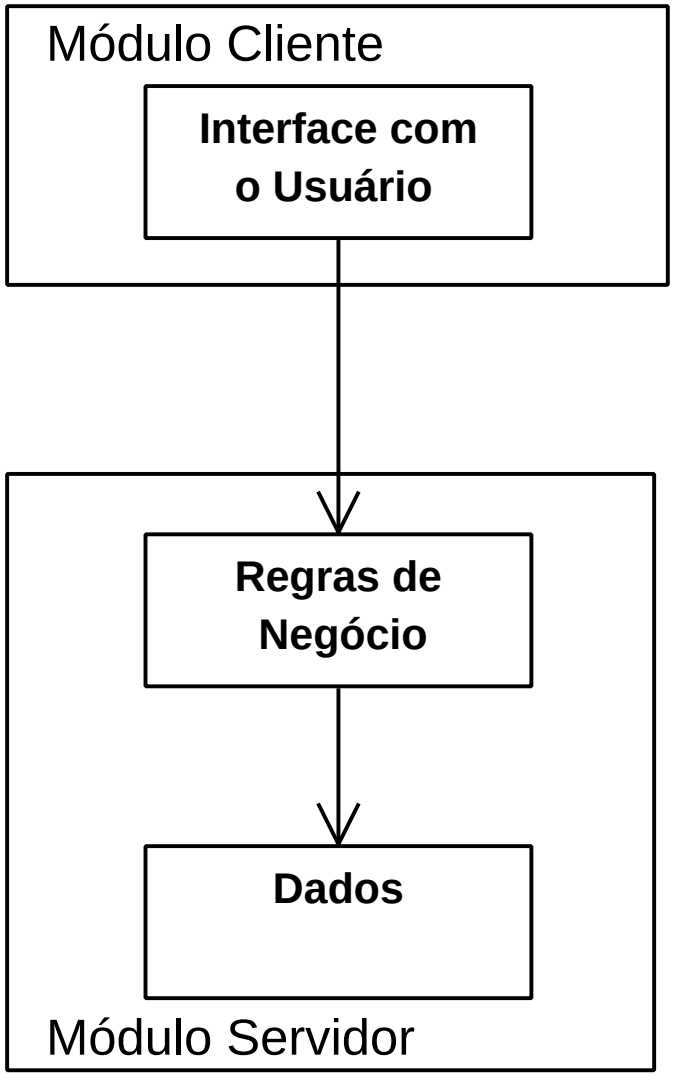
- Tendo em vista os requisitos de distribuição, foram identificados dois módulos principais para o sistema de Caixa Automático:
  1. **Subsistema/Módulo Cliente**: classes Terminal e FronteiraDispensadorNotas.
  2. **Subsistema/Módulo Servidor**: demais classes.

Particionamento do Sistema de Caixa Automático (II)

- O módulo servidor ainda pode ser particionado em outros dois submódulos:
  1. **Submódulo de Negócio**: classes Controlador-Caixa, CaixaEletronico, Conta e DadosCliente.
  2. **SubMódulo de Dados**: classe FronteiraCadastro-Contas.

Camadas do Sistema de Caixa Automático

- Arquitetura em *três camadas*
- Mapeamento direto das camadas de cada subsistema.
- Camadas: **cliente**, **negócios** e **dados**.


Módulo Cliente
Interface com o Usuário
Regras de Negócio
Dados
Módulo Servidor

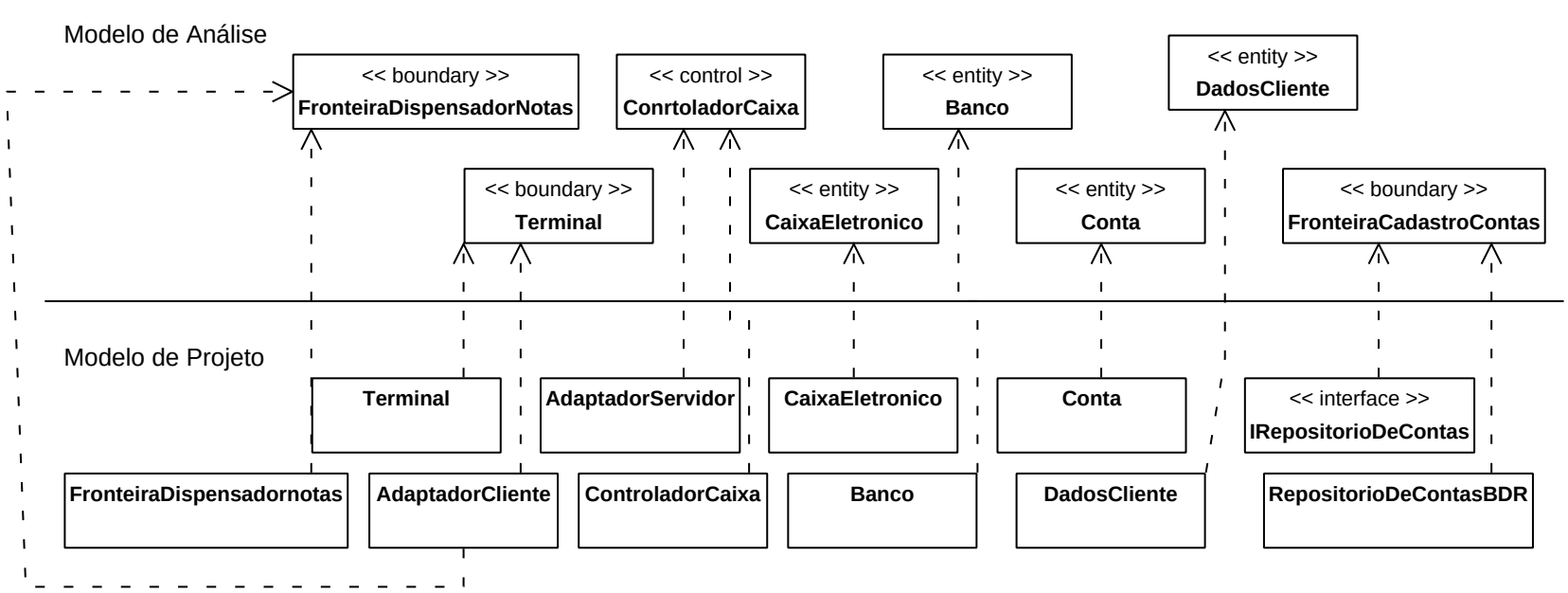
Interfaces entre as Camadas no Sistema de Caixa Automático

- A comunicação entre as camadas de cliente e negócios poderá ser distribuída.
- Classes especiais, chamadas **adaptadores**, serão usadas para encapsular a distribuição.
  - A fim de materializar o aspecto "Cliente-Servidor" do sistema
- As camadas de negócios e dados devem ser desacopladas, de modo que esta última possa ser modificada sem afetar a primeira.

Mapeamento entre as Classes de Análise e as de Projeto (I)

- Após identificar a arquitetura do sistema e as interfaces entre os seus componentes arquiteturais, é possível refinar as classes de análise, a fim de adicionar entidades relativas à implementação do sistema
- Esse refinamento consiste basicamente na adição de novas classes a fim de materializar os requisitos de comunicação e baixo acoplamento entre as camadas da arquitetura

Mapeamento entre as Classes de Análise e as de Projeto (II)



Divisão das Classes de Projeto entre as Camadas da Arquitetura.

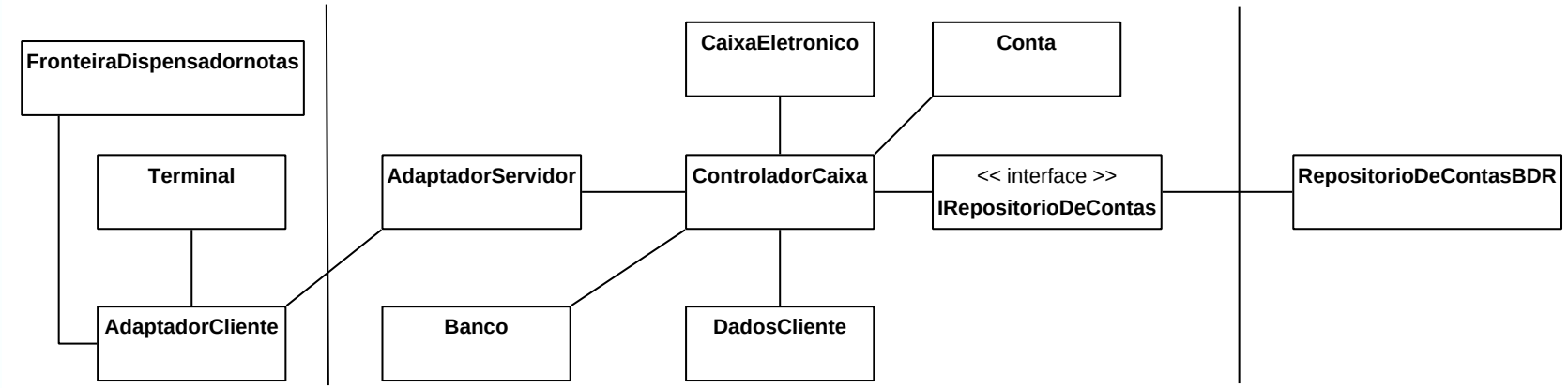


Diagrama de Seqüência da Análise

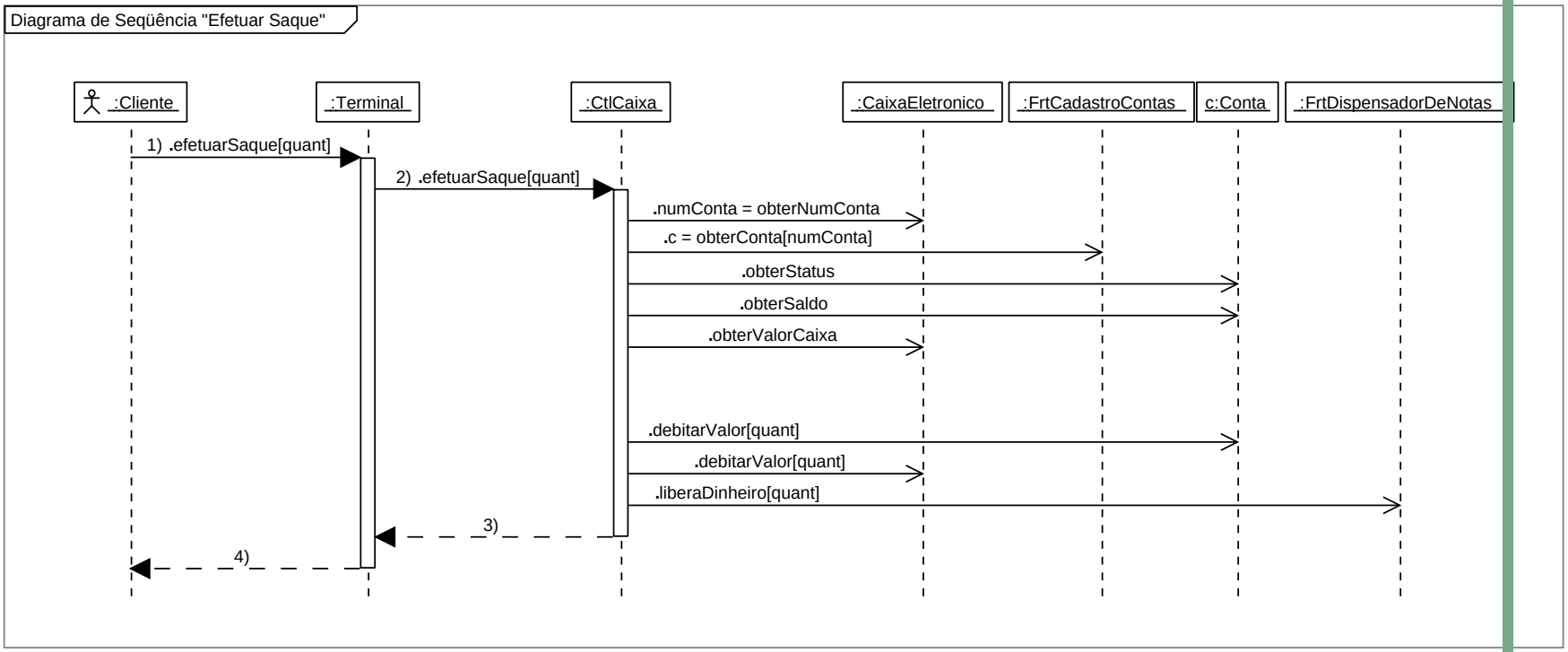
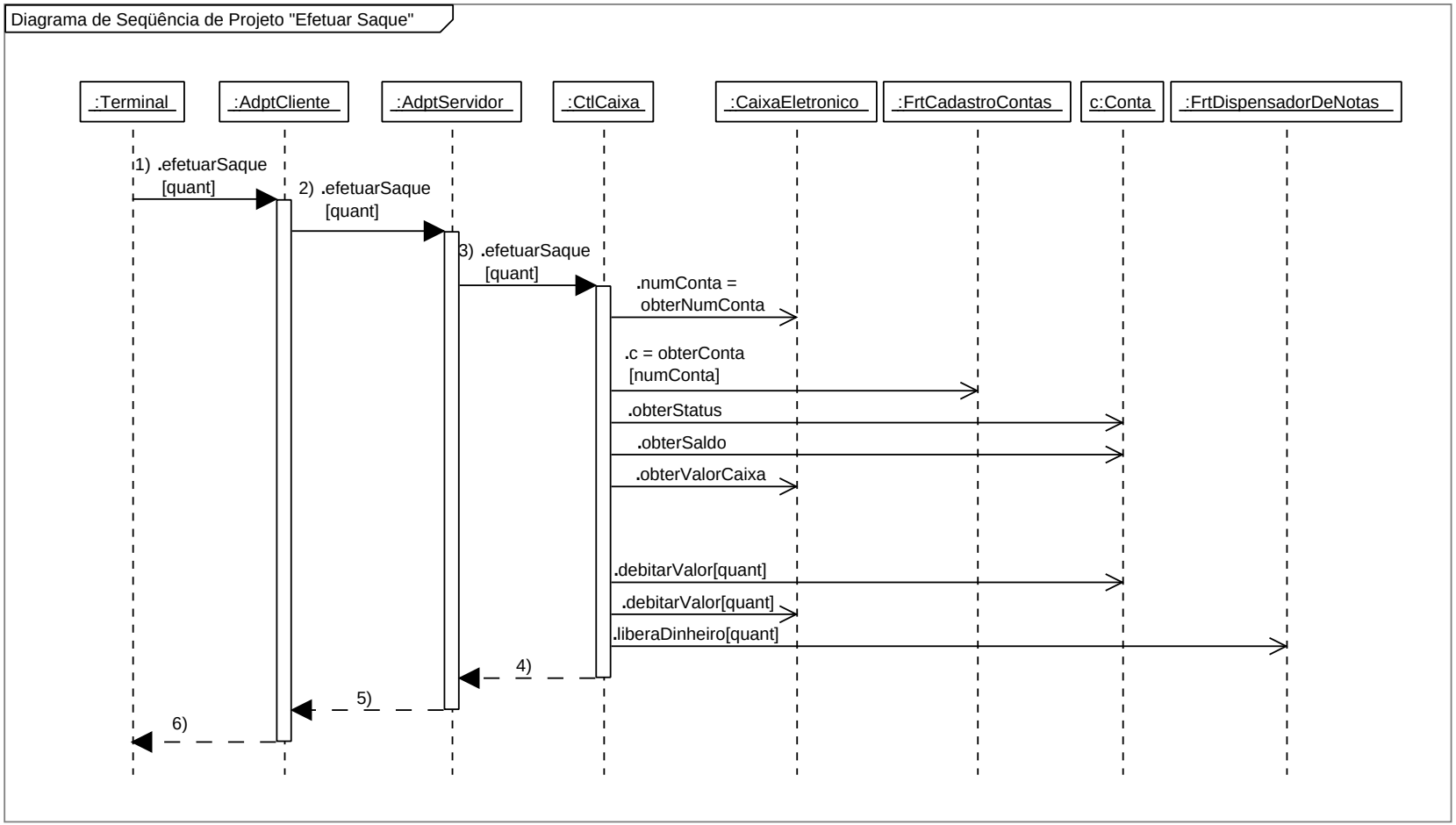


Diagrama de Seqüência do Projeto



Padrões de Projeto

O Padrão de Projeto Singleton

- O padrão singleton busca garantir que uma classe tenha sempre uma única instância que possa ser acessada por vários clientes distintos.
- Esse padrão pode ser utilizado também para oferecer um número fixo de instâncias (2,3,4, etc).

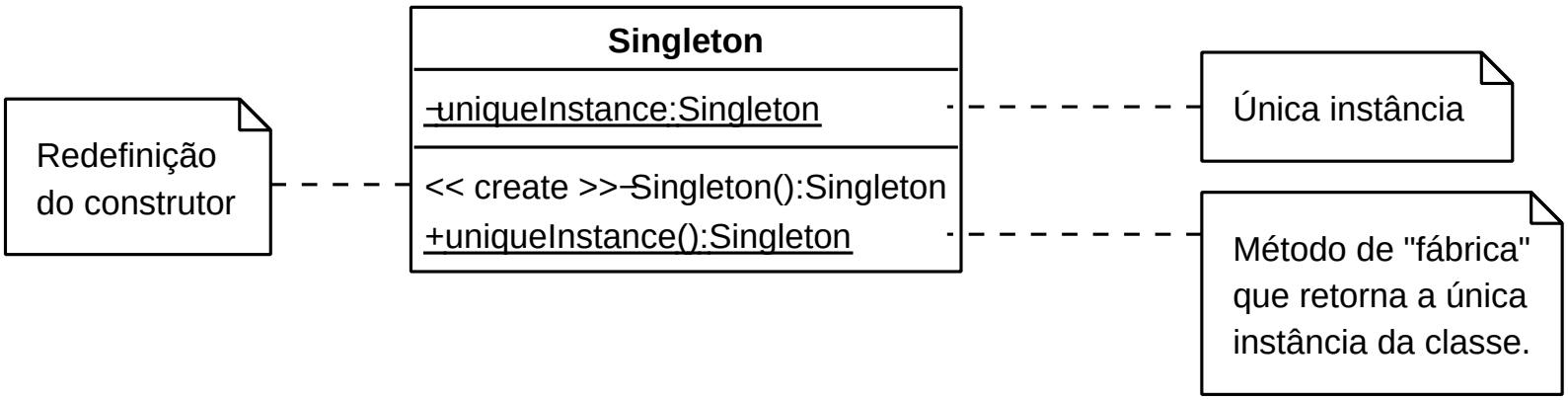
Problema

- Em algumas situações, é importante que as classes garantam a existência de apenas uma única instância criada.
- Normalmente esse problema é resolvido utilizando variáveis globais (uma para cada instância desejada).
- Além de aumentar a complexidade do sistema, essa solução é limitada no sentido de não permitir alterar facilmente o número de instâncias desejadas.

Solução

- Uma solução mais adequada é tornar a classe responsável por si mesma, garantindo que haja somente um número estipulado de instâncias.
- Por se tratar de um acesso centralizado, a classe pode garantir que nenhuma outra instância possa ser criada, através da interceptação de todas as requisições de criação de novas instâncias.
- Dessa forma, é necessário definir uma operação de classe que seja a única responsável pela instanciação dos seus objetos.
- O construtor deve ser definido com visibilidade privada.

Estrutura do Padrão Singleton



Consequências

- **Acesso controlado a um número estipulado de instâncias.** O acesso centralizado possibilita um controle estrito sobre como e quando essas referências são acessadas pelos clientes.
- **Redução do espaço de nomes.** O padrão Singleton apresenta vantagens em relação à utilização de variáveis globais, uma vez que evita a explosão do número dessas variáveis.

Visibilidade

- A implementação do padrão singleton se baseia no ocultamento de informações através do conceito de visibilidade privada.
- Uma classe Singleton deve possuir uma operação de classe ("static") que seja responsável por controlar a instanciação dos seus objetos.
- Além disso, a classe deve necessariamente definir o seu construtir com visibilidade privada, a fim de impossibilitar o uso do construtor diretamente por um cliente.

Exemplo

```
public class ControladorCaixa {
   private static ControladorCaixa instancia;
                  // variável que referencia
                  // a única  instância da classe
   private ControladorCaixa(); //vis. privada do
          //construtor para impedir que a classe
          //seja instanciada diretamente pelos clientes
   public static ControladorCaixa obterInstancia(){
      if (instancia == null) {
         instancia = new ControladorCaixa();
      }
      return instancia;
   }
   ...
}
```

O Padrão de Projeto Composite

- Frequentemente, programadores desenvolvem sistemas onde os componentes podem ser objetos individuais ou podem representar uma coleção.
- As hierarquias de "parte-todo" por si só não são adequadas para a representação de estruturas de árvores, onde alguns nós são simples e outros são compostos.
- Em resumo, uma composição é uma coleção de objetos, podendo ser ums estruturação recursiva, onde uma composição é composta de outras composições.

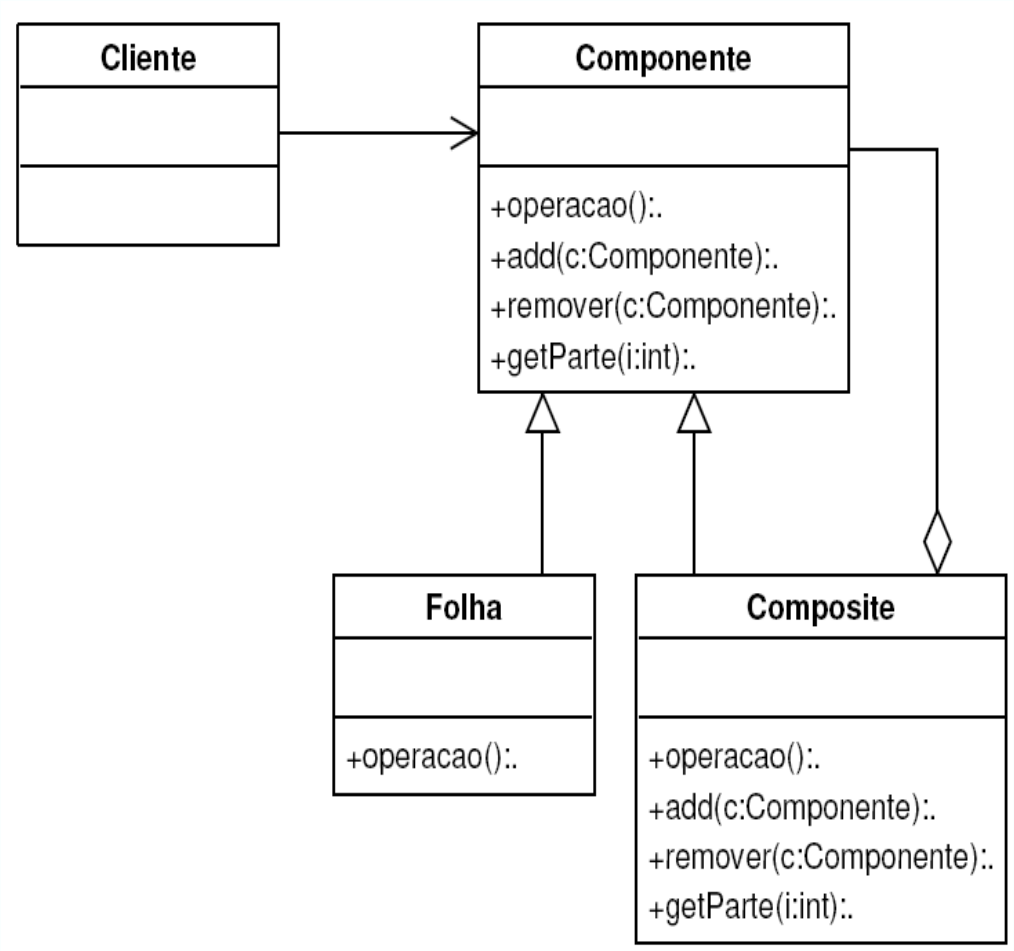
Problema

- O problema principal solucionado por esse padrão é a dicotomia entre ter um única interface para acessar todos os elementos de uma composição, sejam eles simples ou compostos.
- Além dessa interface única, os objetos compostos devem propagar informações para os seus "filhos".
- O terceiro problema enfrentado é o aspecto dinâmico da composição, isto é, um objeto simples pode passar a ser composto em um determinado ponto da execução.

Solução

- Alguns autores sugerem a criação de interfaces para o gerenciamento das composições. Essas interfaces devem possibilitar a adição e remoção de "partes" nas classes.
- Além disso, um objeto deve possibilitar o acesso e a listagem de suas partes (ou filhos).

Estrutura do Padrão Composite



Consequências (I)

- **Define dinamicamente uma hierarquia de classes formada de objetos primitivos e compostos.** Objetos primitivos podem se tornar compostos ao adicionar uma "parte".
- **Simplifica a implementação.** Clientes podem tratar estruturas compostas e simples de uma maneira uniforme.

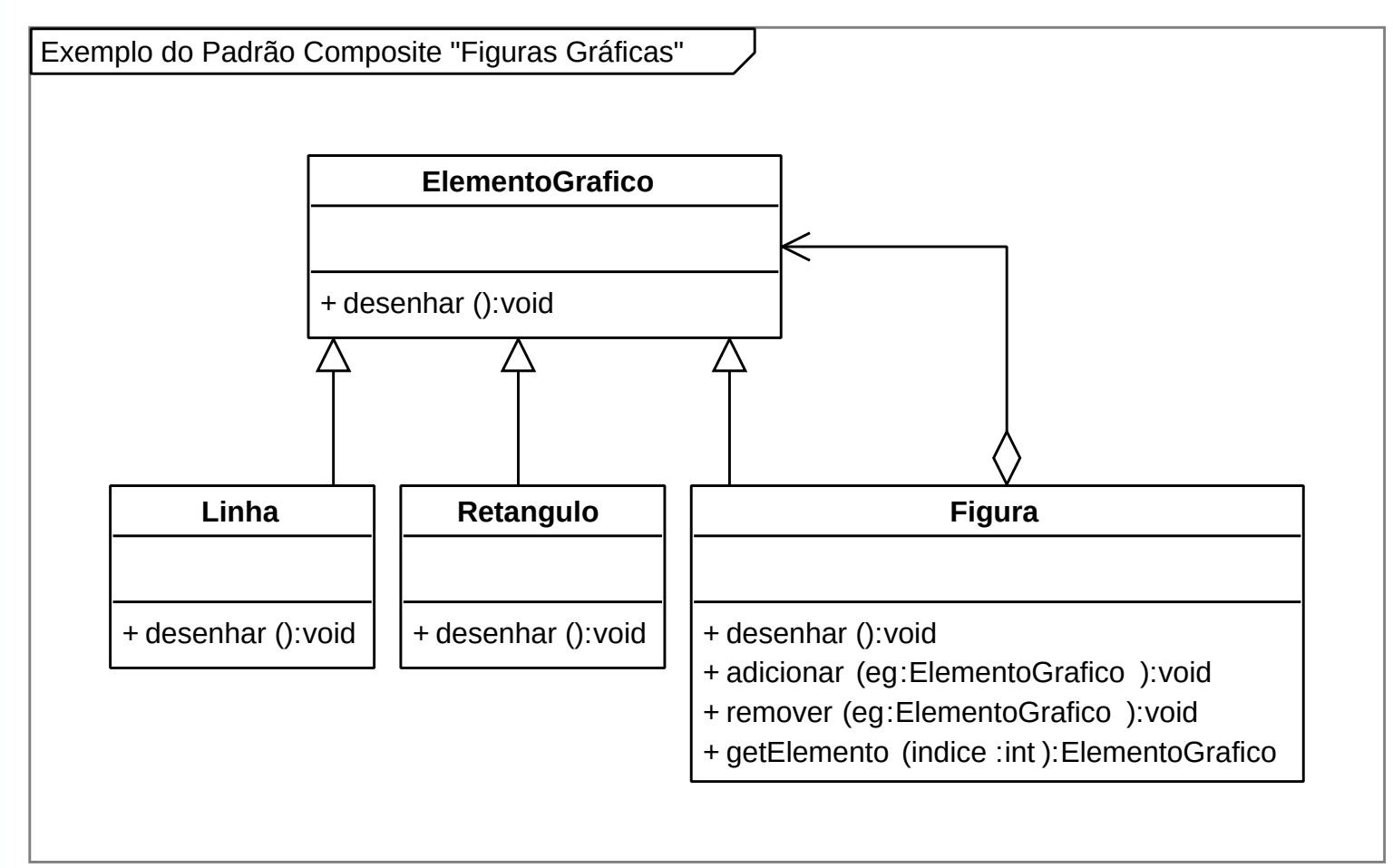
Consequências (II)

- **Facilita a evolução do sistema.** Não são necessárias mudanças drásticas para utilizar classes simples já existentes como componentes de outras classes.
- **Torna o projeto "genérico".** A principal desvantagem desse padrão é a dificuldade em restringir os tipos de classes que podem fazer parte das classes componentes.

Figuras Gráficas (I)

- Nesse sistema, um elemento gráfico pode ser uma linha, um retângulo ou uma figura.
- Porém, uma figura pode ser composta de vários elementos gráficos, inclusive outras figuras.
- Esse é um caso típico de uso do padrão Composite.

Figuras Gráficas (II)



Implementação (I)

- **Referência explícita entre o "todo" e suas "partes"**. Essas referências facilitam o gerenciamento da estrutura.
- **Compartilhamento de componentes.** O mesmo objeto pode jazer parte de mais de um outro.
- **Deve-se maximizar a interface comum dos componentes**. A classe componente deve definir o máximo de operações comuns.

Implementação (II)

```
public abstract class ElementoGrafico {
  private String cor;

  public ElementoGrafico(String cor){
     this.cor = cor;
  }

  public abstract void desenhar();
}
```

Implementação (III)

```
public class Linha extends ElementoGrafico{
  private int[] ponto1;
  private int[] ponto2;

  public Linha(String cor, int[] ponto1, int[] ponto2){
    super(cor);
    this.ponto1 = ponto1;
    this.ponto2 = ponto2;
  }

  public void desenhar(){
    //... desenhar a reta ‘‘ponto1 - ponto2’’
  }
}
```

Implementação (IV)

```
public class Retangulo extends ElementoGrafico{
  private int[] centro;
  private int lado1;
  private int lado2;

  public Retangulo(String cor,
           int[] centro, int lado1, int lado2){
    super(cor);
    this.centro = centro;
    this.lado1 = lado1;
    this.lado2 = lado2;
  }
  public void desenhar(){
    //desenhar um retângulo
  }}
```

Implementação (V)

```
public class Figura extends ElementoGrafico{
   private ElementoGrafico[] elementos;
  public Figura(String cor){
    super(cor);
  }

  public void desenhar(){
    para cada elemento{
      elementoAtual.desenhar();}
  }
  public void adicionar(ElementoGrafico eg){...}
  public void remover(ElementoGrafico eg){...}
  public ElementoGrafico getElemento(int indice){
     return elemento[indice];}
}
```

O Padrão de Projeto Observer

- O Padrão Observer é utilizado para atualizar a exibição de dados em diferentes visualizações.
- Mais especificamente, imaginemos uma situação onde um dado é exibido em forma de planilha e gráfico (simultaneamente).
- É desejável que as duas visualizações se mantenham consistentes durante a execução do sistema.

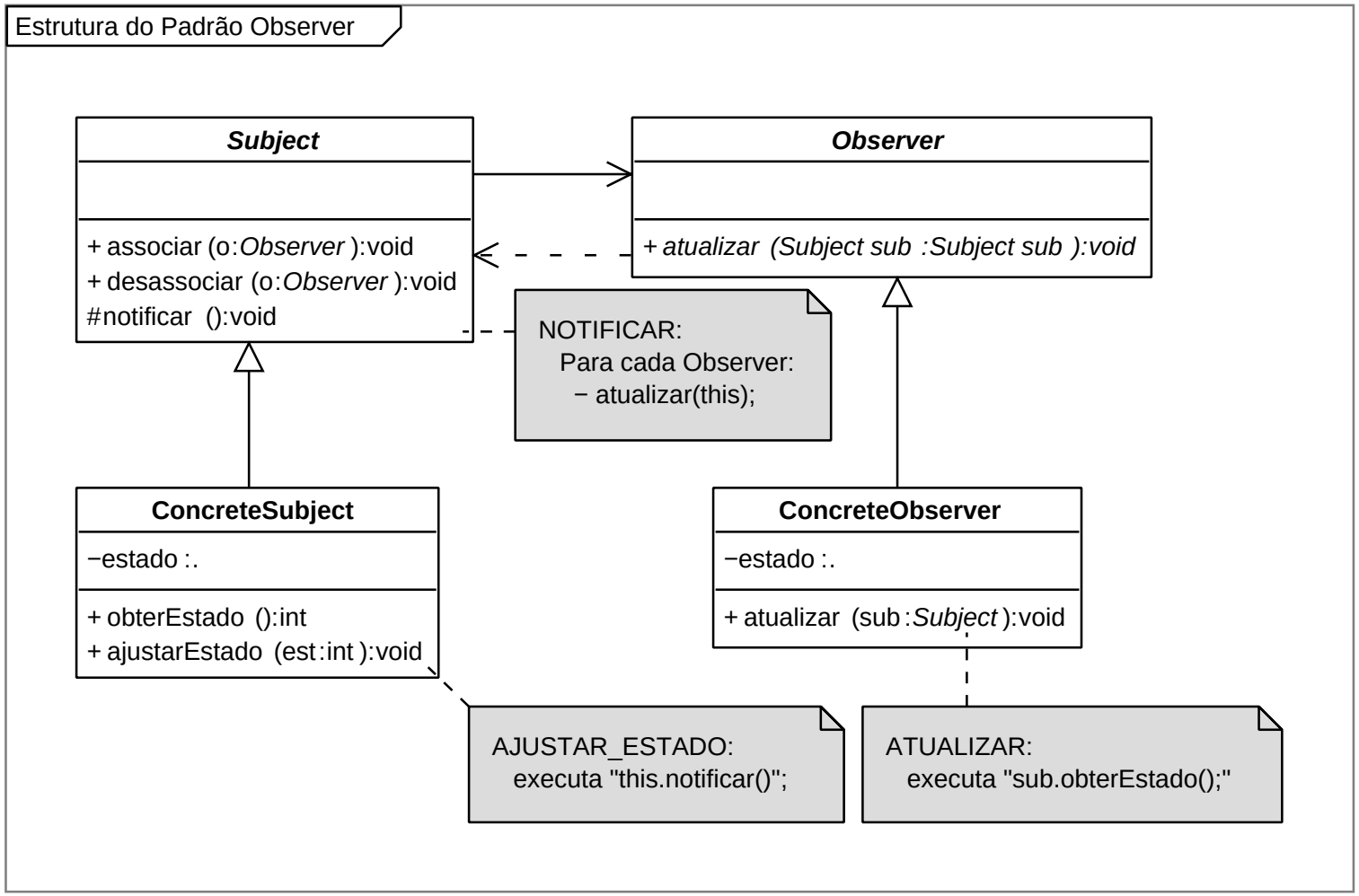
Problema

- Atualmente, com o advento das janelas gráficas, é cada vez mais comum a neessidade de atualizações múltiplas de exibições que reflitam os mesmos dados.
- Por exemplo, pode ser necessário reresentar os dados de uma planilha como um gráfico de barras e como um gráfico de pizza, além da própria tabela. Quando atualizar a planilha, é desejável que os gráficos atualizem automaticamente.

Solução

- Ao implementar o padrão observer, normalmente referimos aos dados como o sujeito da ação e as suas formas de exibição como observadores (do inglês observers).
- Cada observador se registra com seus respectivos objetos de interesse a fim de serem notificados de alterações nos seus estados.
- Dessa forma, cada observador deve conhecer a interface que o sujeito oferece para cadastro e para comunicação de mudanças.

Estrutura do Padrão Observer



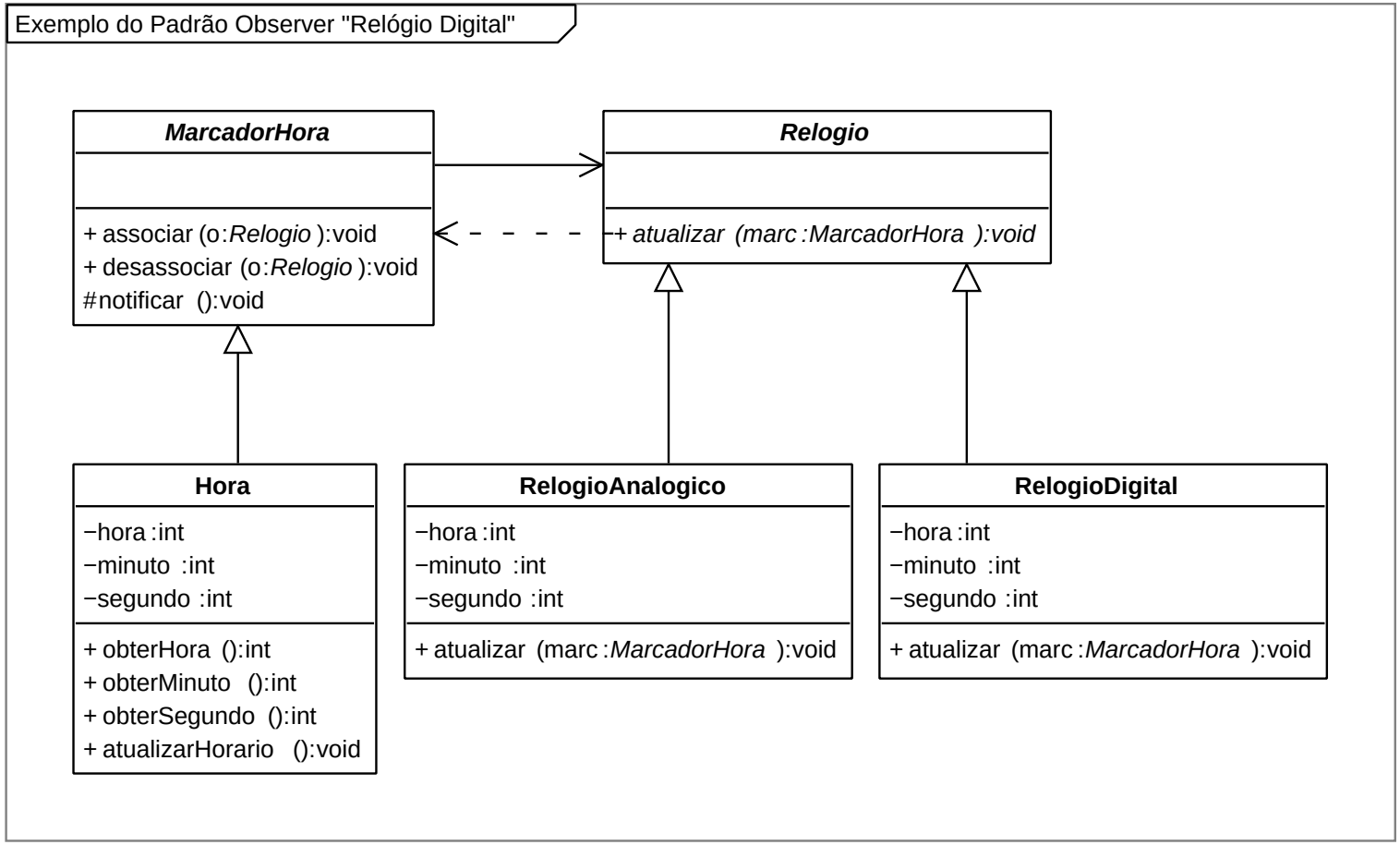
Consequências

- **Acoplamento mínimo entre o sujeito e os observadores.** Tudo que o sujeio sabe é que possui uma lista de observadores que oferecem uma interface simples. Dessa forma, os sujeitos não conhecem o tipo concreto que materializa o observador.
- **Suporte a disseminação de mensagens (broadcasting).** A notificação que o sujeito envia é automaticamente submetida para todos os seus observadores.
- **Atualizações inesperadas.** Devido aos observadores não se conhecerem, a mudança coletiva se torna "transparente", bastando para isso que eles mudem o sujeito.

Exemplo de Uso - Relógio Digital (I)

- Nesse exemplo, as horas de um relógio pode ser representada de forma analógica e digital.
- Essas duas visualizações devem estar sincronizadas com o valor real da hora (hora, minuto e segundo).
- Dessa forma, a mudança nos dados da hora devem influenciar as duas visualizações, cada uma representando da sua maneira.

Exemplo de Uso - Relógio Digital (II)



Implementação (I)

- Os observadores pecisam conhecer o seu sujeito.
- Pode ser necessário que um observador possua mais de um sujeito.
- O sujeito e seus observadores utilizam o mecanismo de notificação para manter a consistência.
- A remoção de um sujeito não deve produzir referências inconsistentes nos seus observadores. Para isso, o sujeito deve notificar seus observadores antes de ser removido.

Implementação (II)

```
public class MarcadorHora {
  private ‘‘Vetor de Relogio’’ observers;

  public void associar (Relogio r){
    observers.‘‘addLista(r)’’;
  }
  public void desassociar (Relogio r){
    observers.‘‘removerLista(r)’’;
  }
  void notificar() {
    ‘‘Para cada observer’’(observers){
      itemCorrente.atualizar(this);} //delegação
  }
}
```

Implementação (III)

```
public class Hora extends MarcadorHora {
  private int hora;
  private int minuto;
  private int segundo;

  public Hora(int hora, int minuto, int segundo){
    this.hora = hora;
    this.minuto = minuto;
    this.segundo = segundo;
  }
  ...obterHora(); obterMinuto(); obterSegundo()}
public class Relogio {
  public abstract void atualizar(MarcadorHora marc);
}
```

Implementação (IV)

```
public class RelogioAnalogico extends Relogio {
  private int hora; private int minuto;
  private int segundo;

  public RelogioAnalogico(MarcadorHora marc){
   this.hora = marc.obterHora();
   this.minuto = marc.obterMinuto();
   this.segundo = marc.obterSegundo();
   marc.adicionar(this);
  }
  public void atualizar(MarcadorHora s){
        // redefinição que utiliza ‘‘marc.obter*()’’
  }
  public void mostrarHora(){//mostrar hora analogico}
}
```

Implementação (V)

```
public class RelogioDigital extends Relogio {
  private int hora; private int minuto;
  private int segundo;

  public RelogioDigital(MarcadorHora marc){
   this.hora = marc.obterHora();
   this.minuto = marc.obterMinuto();
   this.segundo = marc.obterSegundo();
   marc.adicionar(this);
  }
  public void atualizar(MarcadorHora s){
        // redefinição que utiliza ‘‘marc.obter*()’’
  }
  public void mostrarHora(){//mostrar hora digital}
}
```

Implementação (VI)

```
public class RelogioAnaDigi{
  private RelogioAnalogico ra;
  private RelogioDigital rd;
  private MarcadorHora marc;

  public Relogio(MarcadorHora marc){
    this.marc = marc;
    this.ra = new RelogioAnalogico(marc);
    this.rd = new RelogioDigital(marc);
  }
  public void mostrarHora(){
    ra.mostrarHora();
    rd.mostrarHora();
  }
  ...
```

Implementação (VI)

```
  ...
  public static void main(String args[]){
    MarcadorHora marc = new Hora(12, 30, 00);
    RelogioAnaDigi rel = new RelogioAnaDigi(marc);
    rel.mostrarHora();
  }
}
```