Uma Introdução aos Padrões de Projeto com Java

Roberto Willrich
INE-CTC-UFSC

Introdução aos Padrões de Projeto

- Programação
  - Introdução
    - Motivação, Definição, Características, Histórico
  - Descrição de um padrão
  - Classificações dos Padrões
  - Catalogo de Padrões GoF
  - Exemplos de Padrões de Projeto

Introdução: Motivação

- Projeto de software, orientados-a-objetos, é uma tarefa complexa
  - Simples uso da OO não garante que obtenhamos sistemas confiáveis, robustos, extensíveis e reutilizáveis.
- É difícil achar algo reusável na primeira tentativa
  - **Reusabilidade** real não se obtém de técnicas de "copiar & colar" nem do simples reaproveitamento de módulos de software.
    - É preciso encontrar objetos pertinentes, fatorá-los em classes na granularidade certa, definir interfaces de classes e hierarquias de herança, e estabelecer relacionamentos chave entre elas;
- Ainda assim boas soluções são produzidas:
  - Boas soluções que já funcionaram devem ser reutilizadas (reuso de experiências anteriores);

Introdução: Motivação

- Existem classes de problemas que se repetem em uma diversidade de domínios
  - Consequentemente: soluções se repetem!
- Padrões de Projeto: descrevem problemas recorrentes no projeto de sistemas e sua solução em termos de interfaces e objetos
  - Nome, problema, solução, consequências...
  - É reusar projetos e arquiteturas de sucesso, ou seja, técnicas comprovadas, em forma de um catálogo, num formato consistente e acessível para projetistas;

Introdução

- Definições de Padrões de Projeto
  - *"Cada padrão descreve um problema que se repete várias vezes em um determinado meio, e em seguida descreve o âmago da sua solução, de modo que esta solução possa ser usada milhares e milhares de vezes"*

    ALEXANDER, C., **ISHIKAWA**, S., **SILVERSTEIN**, M., **JACOBSON**, M., **FIKSDAHL-KING**, I., **ANGEL**, S. "*A Pattern Language*". New York, NY (USA): Oxford University Press, 1977.

  - *"Um padrão de projeto sistematicamente nomeia, motiva e explica um projeto genérico, que endereça um problema de projeto recorrente em sistemas orientados a objetos. Ele descreve o problema, a solução, quando é aplicável e quais as conseqüências de seu uso."*

    GAMMA, E., **HELM**, R., **JOHNSON**, R., **VLISSIDES**, J. "*Design Patterns: Elements of Reusable Object-Oriented Software*". Reading, MA: Addison Wesley, 1995.

Introdução

- Os padrões de projeto beneficiam os desenvolvedores de um sistema
  - Ajudando a construir um software confiável com arquiteturas testada e perícia acumulada pela indústria
  - Promovendo a reutilização de projetos em futuros sistemas
    - Permitem compartilhar experiências bem sucedidas na resolução de problemas recorrentes.
  - Ajudando a identificar equívocos comuns e armadilhas que ocorrem na construção dos sistemas
  - Estabelecendo um vocabulário comum de projeto entre os desenvolvedores
  - Encurtando a fase de projeto no processo de desenvolvimento de um software
    - Permitem que os desenvolvedores concentrem seus esforços nos aspectos inéditos do problema.

Catálogo de soluções

- Desde 1995, o desenvolvimento de software passou a ter o seu primeiro catálogo de soluções para projeto de software: o livro GoF.
  - Catálogo GoF (“the gang of four”)
    - “Design Patterns: Elements of Reusable Object-Oriented Software,” Gamma, Helm, Johnson, Vlissides, Addison-Wesley, 1995
  - Passamos a ter um vocabulário comum para conversar sobre projetos de software.
    - Soluções que não tinham nome passam a ter nome.
    - Ao invés de discutirmos um sistema em termos de pilhas, filas, árvores e listas ligadas, passamos a falar de coisas de muito mais alto nível como Fábricas, Fachadas, Observador, Estratégia, etc.

Descrição do padrão: Forma GoF

- Nome e Classificação
- Propósito
- Nome Secundário
- Motivação
- Aplicabilidade
- Estrutura
- Participantes
- Colaborações
- Conseqüências
- Implementação
- Exemplo
- Usos Conhecidos
- Padrões Relacionados

Descrição do padrão

- Nome do padrão e classificação:
  - Passa a fazer parte do vocabulário dos projetistas.
- Propósito:
  - Respostas para as perguntas - O quê o padrão faz? Que tipo de problema ou característica particular de Projeto ele trata?
- Nomes secundário:
  - Conjunto de outros nomes (apelidos) conhecidos para o padrão, se existir algum.
- Motivação:
  - Um cenário que ilustra o problema e como as estruturas de classes e objetos no padrão o resolvem.
- Aplicação:
  - Respostas para as perguntas - Quais são as situações onde este padrão pode ser aplicado? Quais são os exemplos de projetos que ele pode tratar? Como você pode reconhecer estas situações?

Descrição do padrão

- Estrutura:
  - Uma representação gráfica das classes
- Participantes:
  - As classes e/ou objetos que participam no Padrão de Projeto, e suas responsabilidades.
- Colaborações:
  - Como os participantes interagem para cumprir suas responsabilidades.
- Conseqüências:
  - Resultados e efeitos causados pela aplicação do Padrão
  - Respostas para as perguntas - Como o Padrão alcança seus objetivos? Quais são os resultados do uso do Padrão?

Descrição do padrão

- Implementação:
  - Dicas e técnicas que o projetista deve saber, e possíveis armadilhas para as quais ele deve estar preparado.
- Código Exemplo:
  - Fragmentos de código que ilustrem como o Padrão deve ser implementado
- Usos Conhecidos:
  - Exemplos de utilização do Padrão em sistemas já implementados.
- Padrões Relacionados:
  - Lista de todos os Padrão fortemente relacionados ao Padrão em questão e as suas principais diferenças.

Classificações dos Padrões de Projeto

- Podem ser classificados quanto ao:
  - Quanto ao seu escopo:
    - Classes: padrões tratam do relacionamento entre classes e subclasses;
    - Objetos: padrões tratam relacionamentos entre objetos
  - Quanto ao seu propósito:
    - Padrões Criacionais (Creational)
    - Padrões Estruturais (Structural)
    - Padrões Comportamentais (Behavioral)

Classificações dos Padrões de Projeto

- Padrões Criacionais
  - Todos os Padrões Criacionais tratam da melhor maneira como instanciar objetos;
    - "Classe Criadora" especial pode tornar o programa mais flexível e geral.
- Padrões Estruturais
  - Descrevem como classes e objetos podem ser combinados para formar grandes estruturas;
    - Objetos padrões descrevem como objetos podem ser compostos em grandes estruturas utilizando composição ou inclusão de objetos com outros objetos.
- Padrões Comportamentais
  - Descreve padrões de comunicação entre objetos ou classes
    - Caracteriza o modo como classes e objetos interagem e compartilham responsabilidades.

Catálogo de Padrões de Projeto

			Propósito	
		Criacionais	Estruturais	Comportamentais
Escopo	Classe	Factory Method	Adapter (classe)	Interpreter
				Template Method
	Objeto	Abstract Factory	Adapter (object)	Chain of Responsibility
		Builder	Bridge	Command
		Prototype	Composite	Iterator
		Singleton	Decorator	Mediator
			Façade	Memento
			Flyweight	Observer
			Proxy	State
				Strategy
				Visitor

- Veja suas implementações em java em http://www.fluffycat.com/java-design-patterns/

Catálogo de Padrões de Projeto

- Abstract Factory:
  - Provê uma interface para criar uma família de objetos relacionados ou dependentes sem especificar suas classes concretas.
- Adapter:
  - Converte a interface de uma classe em outra, esperada pelo cliente. Permite que classes que antes não poderiam trabalhar juntas, por incompatibilidade de interfaces, possam agora fazê-lo.
- Bridge:
  - Separa uma abstração de sua implementação, de modo que ambas possam variar independentemente.
- Builder:
  - Provê uma interface genérica para a construção incremental de agregações. Um Builder esconde os detalhes de como os componentes são criados, representados e compostos.

Catálogo de Padrões de Projeto

- Chain of Responsibility:
  - Encadeia os objetos receptores e transporta a mensagem através da corrente até que um dos objetos a responda. Assim, separa objetos transmissores dos receptores, dando a chance de mais de um objeto poder tratar a mensagem.
- Command:
  - Encapsula uma mensagem como um objeto, de modo que se possa parametrizar clientes com diferentes mensagens. Separa, então, o criador da mensagem do executor da mesma.
- Composite:
  - Compõe objetos em árvores de agregação (relacionamento parte-todo). O Composite permite que objetos agregados sejam tratados como um único objeto.
- Decorator:
  - Anexa responsabilidades adicionais a um objeto dinâmicamente. Provê uma alternativa flexível para extensão de funcionalidade, sem ter que usar Herança.

Catálogo de Padrões de Projeto

- Facade:
  - Provê uma interface unificada para um conjunto de interfaces em um subsistema. O Facade define uma interface alto nível para facilitar o uso deste subsistema.
- Factory Method:
  - Define uma interface para criação de um objeto, permitindo que as suas subclasses decidam qual classe instanciar. O Factory Method deixa a responsabilidade de instanciação para as subclasses.
- Flyweight:
  - Usa o compartilhamento para dar suporte eficiente a um grande número de objetos com alto nível de granularidade.
- Interpreter:
  - Usado para definição de linguagens. Define representações para gramáticas e abstrações para análise sintática.
- Iterator:
  - Provê um modo de acesso a elementos de um agregado de objetos, seqüencialmente, sem exposição de estruturas internas.

Catálogo de Padrões de Projeto

- Mediator:
  - Desacopla e gerencia as colaborações entre um grupo de objetos. Define um objeto que encapsula as interações dentre desse grupo.
- Memento:
  - Captura e externaliza o estado interno de um objeto (captura um "snapshot"). O Memento não viola o encapsulamento.
- Observer:
  - Provê sincronização, coordenação e consistência entre objetos relacionados.
- Prototype:
  - Especifica os tipos de objetos a serem criados num sistema, usando uma instância protótipo. Cria novos objetos copiando este protótipo.
- Proxy:
  - Provê projeto para um controlador de acesso a um objeto.
- Singleton:
  - Assegura que uma classe tenha apenas uma instância e provê um ponto global de acesso a ela.

Catálogo de Padrões de Projeto

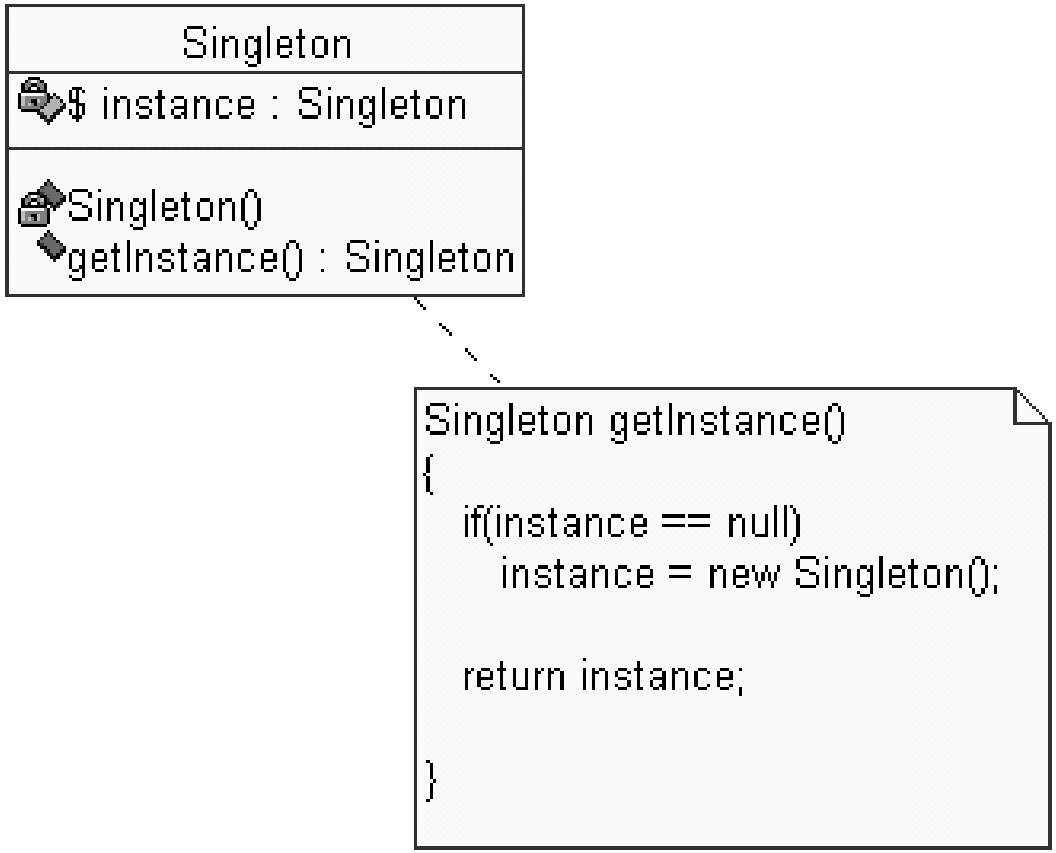
- State:
  - Deixa um objeto mudar seu comportamento quando seu estado interno muda, mudando, efetivamente, a classe do objeto.
- Strategy:
  - Define uma família de algoritmos, encapsula cada um deles, e torna a escolha de qual usar flexível. O Strategy desacopla os algoritmos dos clientes que os usa.
- Template Method:
  - Define o esqueleto de um algoritmo em uma operação. O Template Method permite que subclasses componham o algoritmo e tenham a possibilidade de redefinir certos passos a serem tomados no processo, sem contudo mudá-lo.
- Visitor:
  - Representa uma operação a ser realizada sobre elementos da estrutura de um objeto. O Visitor permite que se crie um nova operação sem que se mude a classe dos elementos sobre as quais ela opera.

Exemplo de Padrão Criacional - Singleton

- Garante que para uma classe específica só possa existir uma única instância, a qual é acessível de forma global e uniforme.
- A classe Singleton deve:
  - armazenar a única instância existente;
  - garante que apenas uma instância será criada;
  - provê acesso a tal instância.
- Exemplos de uso:
  - Um único sistema de arquivos, gerenciador de janelas, tabela de salários (ajuste), leitor de teclado, etc.

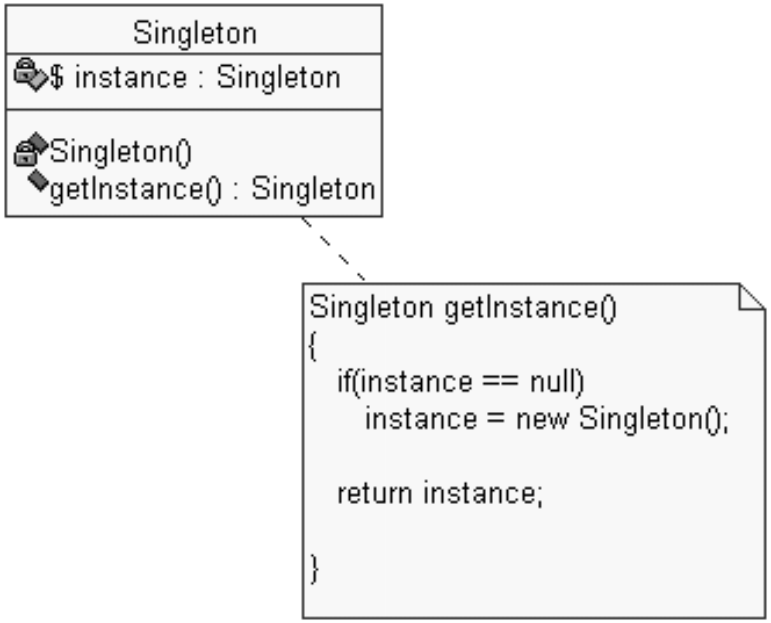
Exemplo de Padrão Criacional - Singleton

- Estrutura:



Exemplo de Padrão Criacional - Singleton

- Implementação
  - Tornar os construtores private de modo que ele não posssam ser chamados externamente
  - Declarar um atributo `private static instance`
  - Escrever o método publico `getInstance()` permitindo acessar a instância unica



Exemplo de Padrão Criacional - Singleton

```
// SingletonImpl.java
public final class SingletonImpl {
   private static SingletonImpl instance = null;
   private SingletonImpl() { ... }
   public static SingletonImpl getInstance() {
      if (instance==null) {
         instance = new SingletonImpl();
      }
      return instance;
   }
}
```

```
public class UsoDoSingletonImpl {
    :
  SingletonImpl obj;
    :
  obj = SingletonImpl.getInstance();
    :
}
```

Exemplo de Padrão Criacional - Singleton

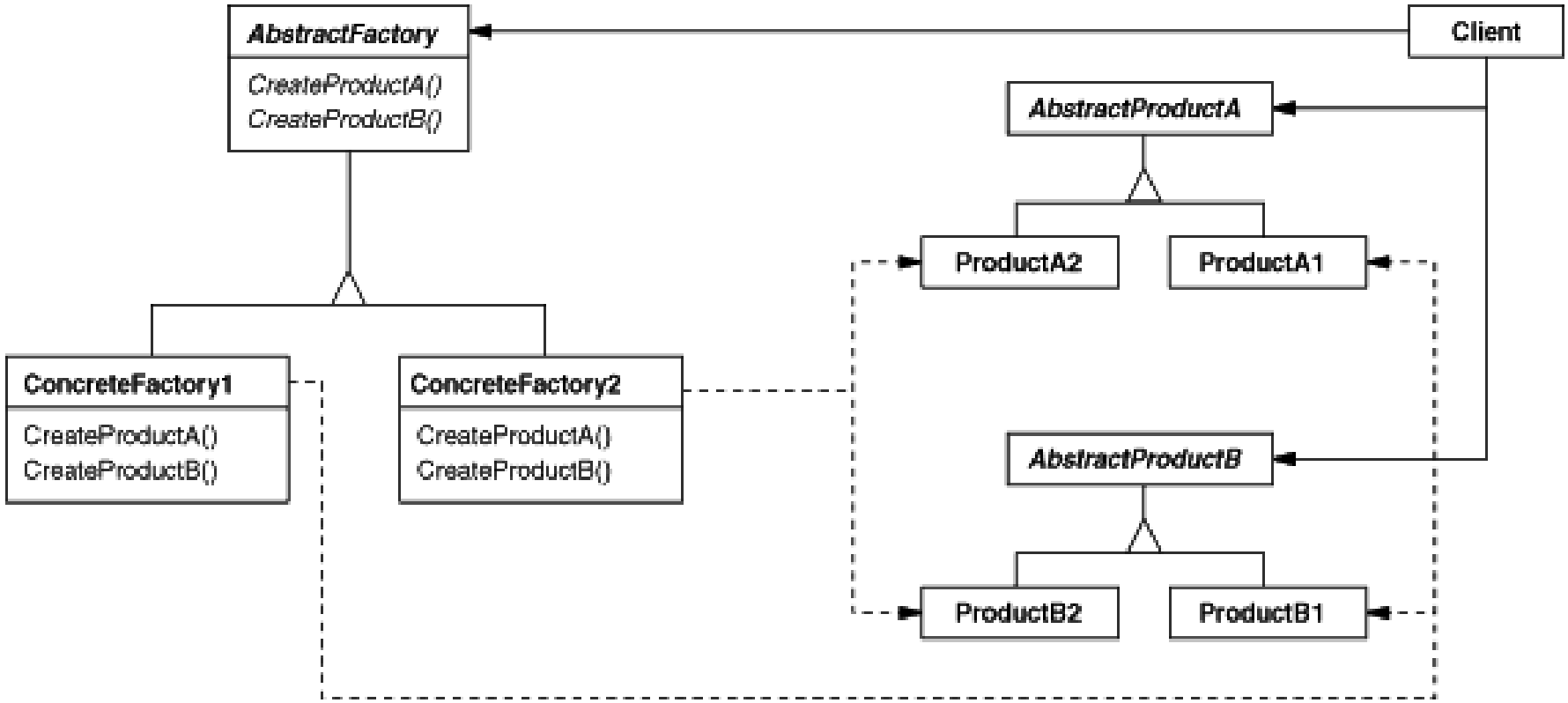
```
public final class RandomGenerator {
   private static RandomGenerator instance = null;
   private RandomGenerator() {}
   public double nextNumber() {
      return Math.random();
   }
   public static RandomGenerator getInstance() {
      if (instance==null) {
         instance = new RandomGenerator();
      }
      return instance;
   }
}
```

Exemplo de Padrão Criacional - Singleton

```
public class TesteRandomGenerator {
    public static void main(String[] args) {
        RandomGenerator gerador;
        gerador = RandomGenerator.getInstance();
        System.out.println("Numero gerado: " + gerador.nextNumber());
    }
}
```

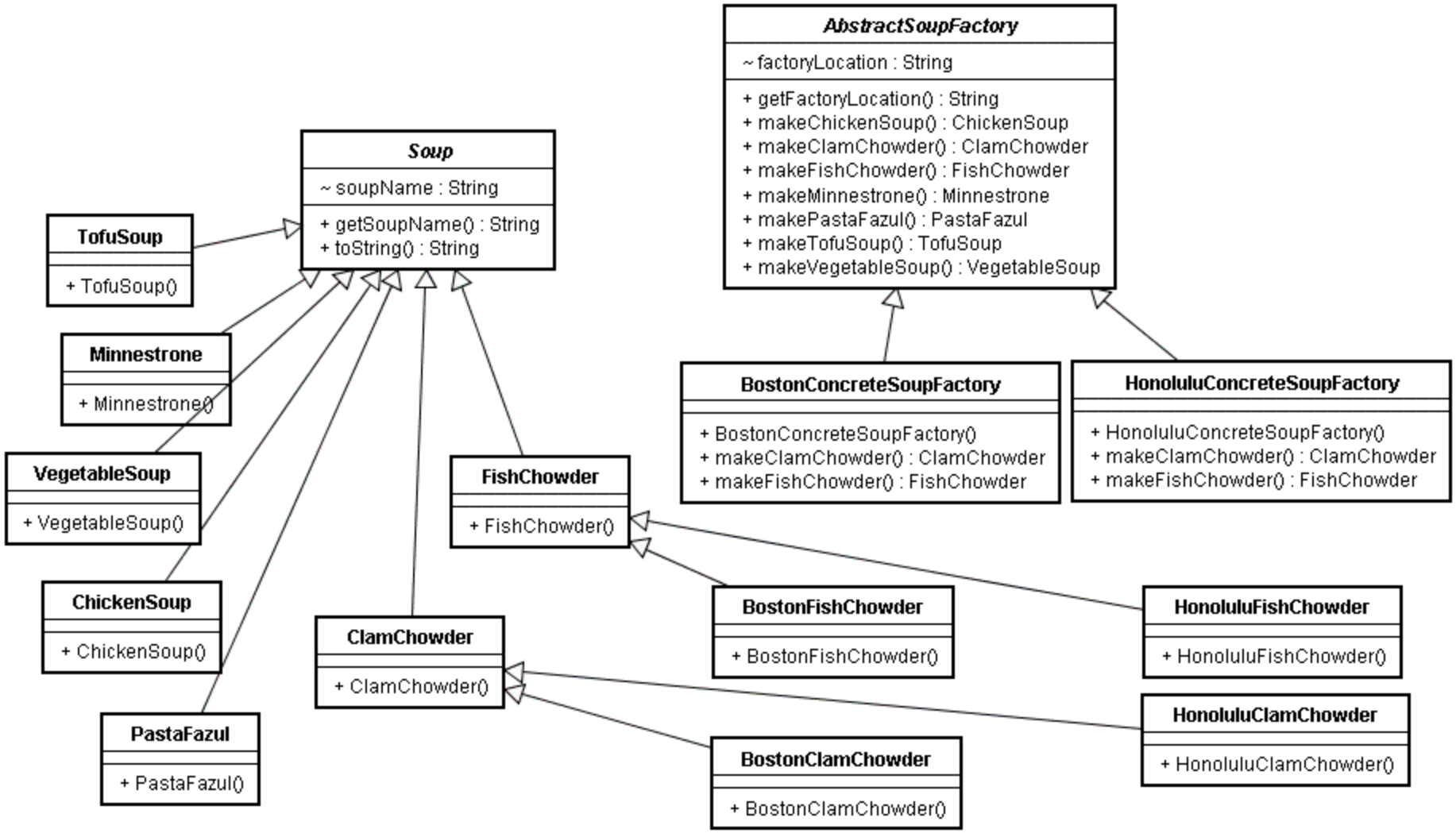
Exemplo de Padrão Criacional – Abstract Factory

- Prover uma interface para criar uma família de objetos relacionados ou dependentes sem especificar suas classes concretas



Exemplo de Padrão Criacional – Abstract Factory

- Exemplo



Exemplo de Padrão Criacional – Abstract Factory

```
abstract class AbstractSoupFactory {
    String factoryLocation;
    public String getFactoryLocation() { return factoryLocation; }
    public ChickenSoup makeChickenSoup() { return new ChickenSoup(); }
    public ClamChowder makeClamChowder() { return new ClamChowder(); }
    public FishChowder makeFishChowder() { return new FishChowder(); }
    public Minnestrone makeMinnestrone() { return new Minnestrone(); }
    public PastaFazul makePastaFazul() { return new PastaFazul(); }
    public TofuSoup makeTofuSoup() { return new TofuSoup(); }
    public VegetableSoup makeVegetableSoup() {return new VegetableSoup(); }
}
```

Exemplo de Padrão Criacional – Abstract Factory

```
class BostonConcreteSoupFactory extends AbstractSoupFactory {
    public BostonConcreteSoupFactory() {
        factoryLocation = "Boston";
    }
    public ClamChowder makeClamChowder() {
        return new BostonClamChowder();
    }
    public FishChowder makeFishChowder() {
        return new BostonFishChowder();
    }
}
```

Exemplo de Padrão Criacional – Abstract Factory

```
class BostonClamChowder extends ClamChowder {
    public BostonClamChowder() {
        soupName = "QuahogChowder";
        soupIngredients.clear();
        soupIngredients.add("1 Pound Fresh Quahogs");
        soupIngredients.add("1 cup corn");
        soupIngredients.add("1/2 cup heavy cream");
        soupIngredients.add("1/4 cup butter");
        soupIngredients.add("1/4 cup potato chips");
    }
}
```

Exemplo de Padrão Criacional – Abstract Factory

```
class BostonFishChowder extends FishChowder {
    public BostonFishChowder() {
        soupName = "ScrodFishChowder";
        soupIngredients.clear();
        soupIngredients.add("1 Pound Fresh Scrod");
        soupIngredients.add("1 cup corn");
        soupIngredients.add("1/2 cup heavy cream");
        soupIngredients.add("1/4 cup butter");
        soupIngredients.add("1/4 cup potato chips");
    }
}
```

Exemplo de Padrão Criacional – Abstract Factory

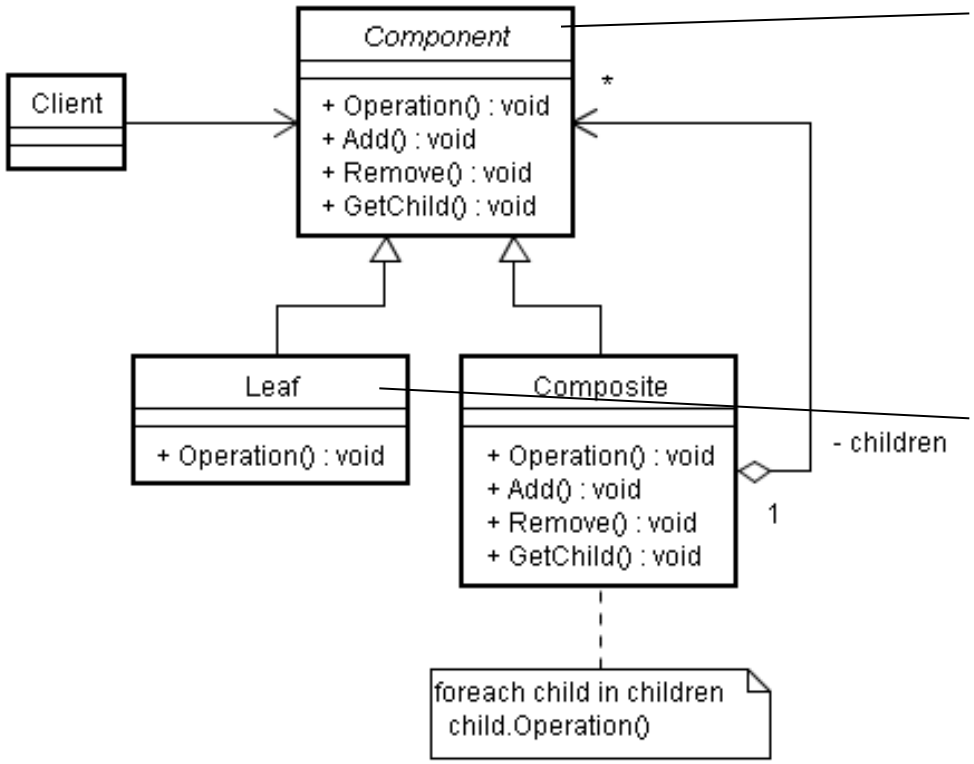
```
import java.util.Calendar;
class TestAbstractSoupFactory {
   public static Soup MakeSoupOfTheDay(AbstractSoupFactory  concreteSoupFactory)  {
       Calendar todayCalendar = Calendar.getInstance();
       int dayOfWeek = todayCalendar.get(Calendar.DAY_OF_WEEK);
       if (dayOfWeek == Calendar.MONDAY) {
           return concreteSoupFactory.makeChickenSoup();
       } else if (dayOfWeek == Calendar.TUESDAY) {
           return concreteSoupFactory.makeClamChowder();
       } else if (dayOfWeek == Calendar.WEDNESDAY) {
           return concreteSoupFactory.makeFishChowder();
       } else if (dayOfWeek == Calendar.THURSDAY) {
           return concreteSoupFactory.makeMinnestrone();
       } else if (dayOfWeek == Calendar.TUESDAY) {
           return concreteSoupFactory.makePastaFazul();
       } else if (dayOfWeek == Calendar.WEDNESDAY) {
           return concreteSoupFactory.makeTofuSoup();
       } else { return concreteSoupFactory.makeVegetableSoup();  }
   }
```

Exemplo de Padrão Criacional – Abstract Factory

```
    public static void main(String[] args)  {
        AbstractSoupFactory concreteSoupFactory = new BostonConcreteSoupFactory();
        Soup soupOfTheDay =  MakeSoupOfTheDay(concreteSoupFactory);
        System.out.println("The Soup of the day " +
                           concreteSoupFactory.getFactoryLocation() +
                           " is " +
                           soupOfTheDay.getSoupName());
        concreteSoupFactory = new HonoluluConcreteSoupFactory();
        soupOfTheDay = MakeSoupOfTheDay(concreteSoupFactory);
        System.out.println("The Soup of the day " +
                           concreteSoupFactory.getFactoryLocation() +
                           " is " +
                           soupOfTheDay.getSoupName());
    }
}
```

Exemplo de Padrão Estrutural - Composite

- Compõe objetos em estrutura de árvore para representar hierarquias do tipo todo-parte.
  - Este padrão permite que as classes cliente tratem os objetos individuais e as composições de maneira uniforme.



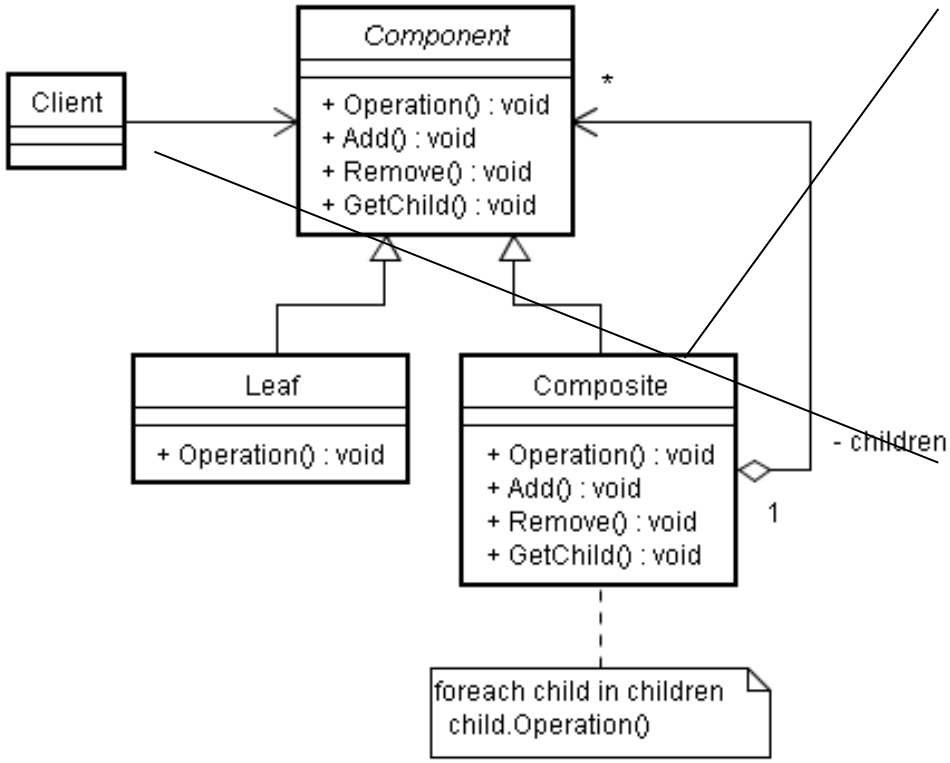
» Declara a interface para objetos da composição.
» Implementa o comportamento padrão comum a todas as classes.
» Declara uma interface para acessar e gerenciar os seus elementos filhos.

» Representa objetos folha na composição, que são aqueles que não possui filhos.

» Define comportamente para objetos primitivos da composição.

Exemplo de Padrão Estrutural - Composite

- Compõe objetos em estrutura de árvore para representar hierarquias do tipo todo-parte.
  - Este padrão permite que as classes cliente tratem os objetos individuais e as composições de maneira uniforme.



» Define o comportamento para componentes que possuem filhos.

» Armazena componentes filhos.

» Implementa as operações relacionadas aos filhos que foram definidas na interface de **Component**.

Manipula objetos da composição através da interface **Component**.

Exemplo de Padrão Estrutural - Composite

```
import java.util.*;
abstract class Component {
    //propriedades
    protected String name ;
    //constructors
    public Component( String name ) {
        this.name = name ;
    }
    //métodos
    abstract public void add ( Component c );
    abstract public void remove ( Component c );
    abstract public void display( int depth );
}
```

Exemplo de Padrão Estrutural - Composite

```
class Composite extends Component {
      //propriedades
      private Vector<Component> children = new Vector<Component>() ;
      //construtor
      public Composite(String name) {
            super(name);  }
      //metodos
      public void add(Component c ) {
            children.add( c ) ;
      }
      public void remove(Component c ) {
            children.remove( c ) ;  }
      public void display( int depth ) {
            System.out.println( Util.separadorNivel( depth ) + name ) ;
            for ( Component child: children)
                  child.display( depth + 2 );
      }
}
```

Exemplo de Padrão Estrutural - Composite

```
class Leaf extends Component {
    public Leaf(String name) {
        super(name);
    }
    public void add(Component c) {
        System.out.println( "Cannot add to a leaf" ) ;
    }
    public void remove(Component c) {
        System.out.println( "Cannot remove from a leaf" ) ;
    }
    public void display(int depth) {
        System.out.println( Util.separadorNivel( depth ) + name ) ;
    }
}
```

Exemplo de Padrão Estrutural - Composite

```
class Util {
    // método auxiliar para criar a string que separa a hierarquia dos
    // objetos
    public static String separadorNivel( int depth) {
            String s = "" ;
            for ( int i = 0 ; i < depth ; i++ ) {
                    s = s + "-" ;
            }
            return s ;
    }
}
```

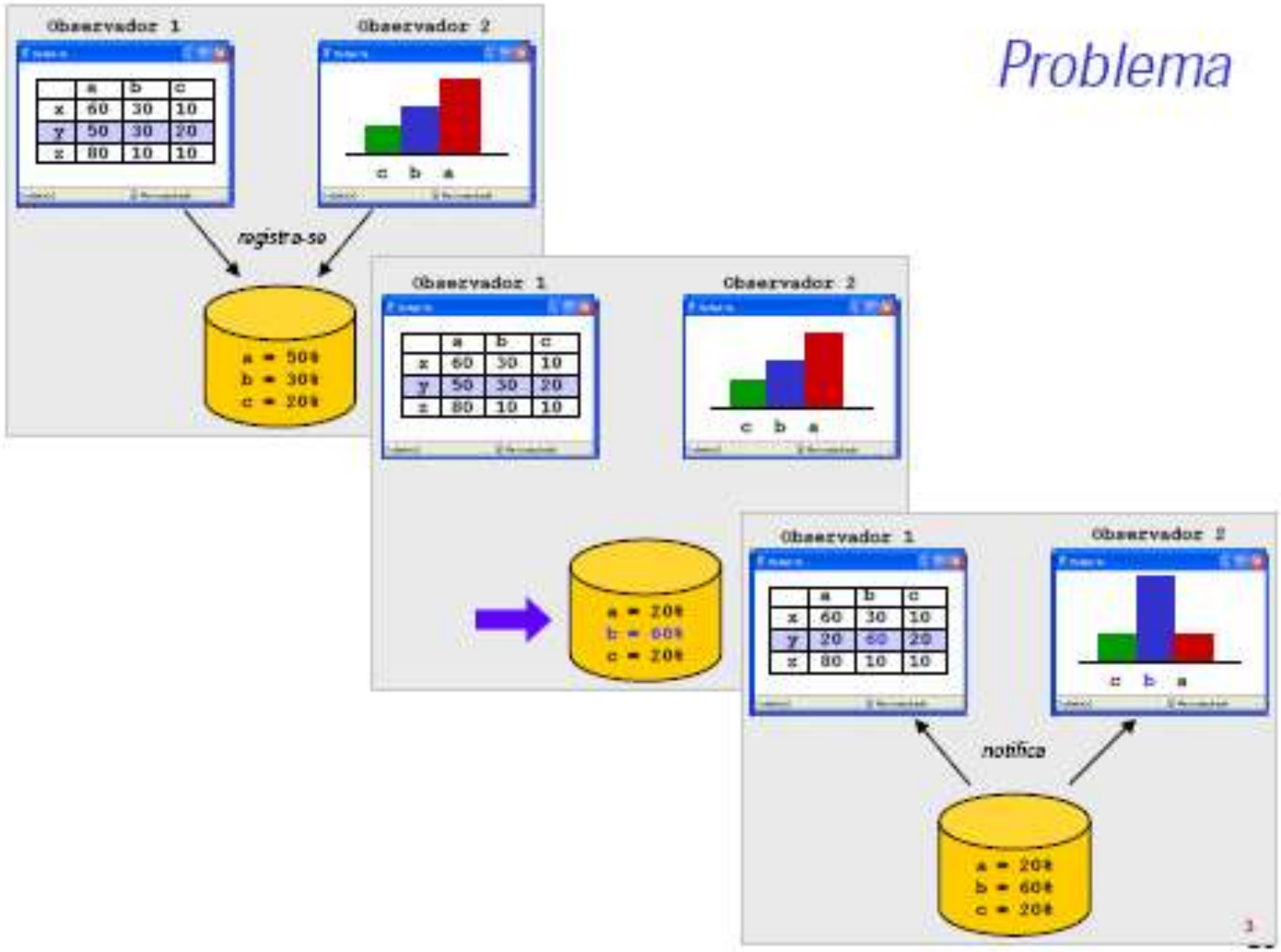
Exemplo de Padrão Estrutural - Composite

```
public class Composite_Estrutural {
    public static void main(String[] args) {
        //cria a estrutura da árvore
        Composite root = new Composite( "root" ) ;
        root.add( new Leaf( "Leaf A" ) ) ;
        root.add( new Leaf( "Leaf B" ) ) ;
        Composite comp = new Composite( "Composite X" ) ;
        comp.add( new Leaf( "Leaf XA" ) ) ;
        comp.add( new Leaf( "Leaf XB" ) ) ;
        root.add( comp ) ;
        root.add( new Leaf( "Leaf C ") ) ;
        //adiciona e remove um Leaf
        Leaf l = new Leaf( "Leaf D" ) ;
        root.add( l ) ;
        root.remove( l ) ;
        //exibe recursivamente os nós
        root.display( 1 ) ;
    }
}
```

Exemplo de Padrão Comportamental - Observer

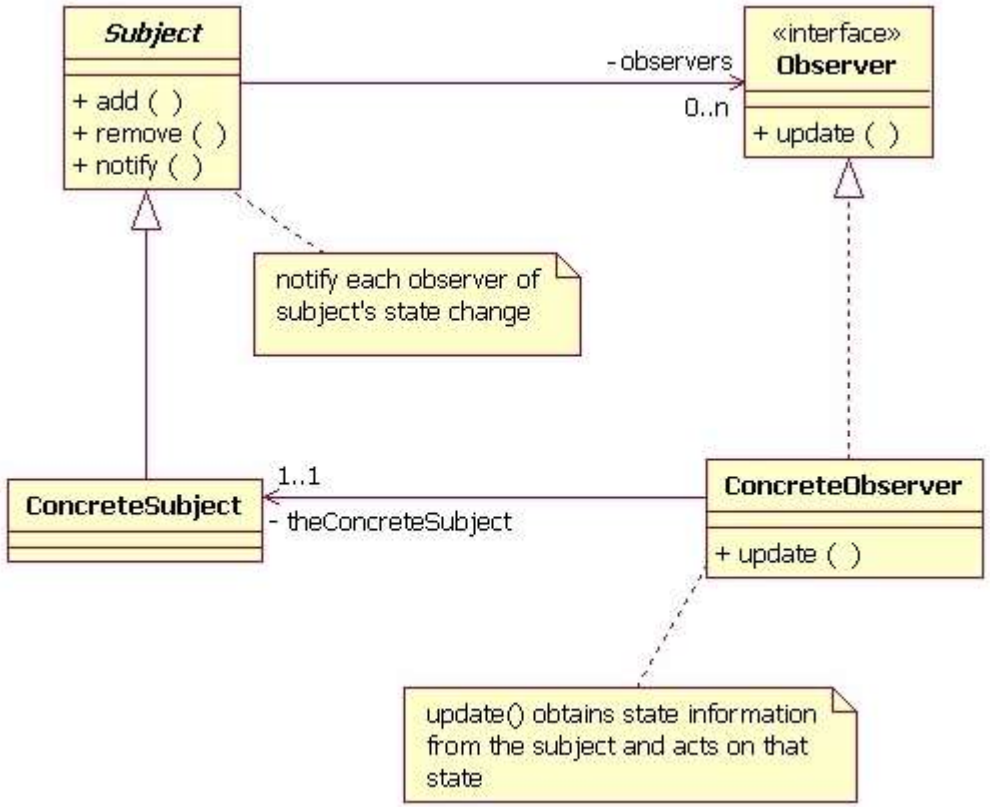
- Existem situações onde diversos objetos (p.e. visualizações) devem representar um outro objeto (p.e. dados).
- Define uma relação de dependência 1:N de forma que quando um certo objeto (assunto) tem seu estado modificado os demais (observadores) são notificados.
- Possibilita baixo acoplamento entre os objetos observadores e o assunto.

Exemplo de Padrão Comportamental - Observer



Exemplo de Padrão Comportamental - Observer

- Estrutura



Exemplo de Padrão Comportamental - Observer

- Java 2 SDK
  - Oferece uma implementação clássica do padrão Observer com a interface Observer e a classe Observable de java.util
    - Classe Observable representa o assunto (subject)
    - Observers implementam a interface Observer
  - Pouco usado pois requer que os assuntos estendam a classe Observable
    - Herança é ruim pois potencialmente os objetos assunto já tem uma superclasse e Java não permite herança múltipla
  - Implementações do padrão observer baseado em eventos é mais interessante
    - Assuntos devem seguir uma convenção que requer os métodos de registro de observação de eventos (listener registration):
      - void addXXXListener(XXXListener)
      - void removeXXXListener(XXXListener)