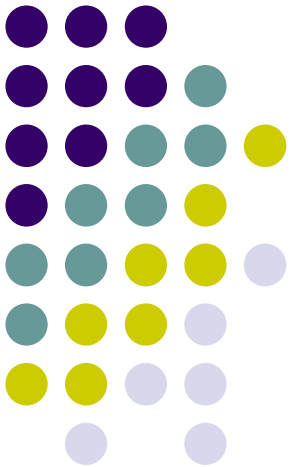
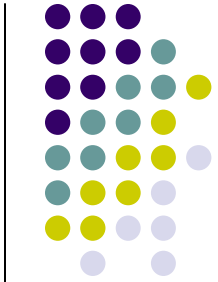
Padrões de Projeto GoF

Prof. Alberto Costa Neto
DComp/UFS

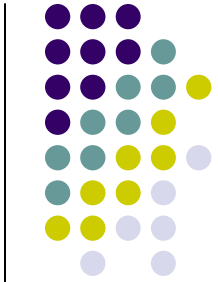


Categorização pelo propósito



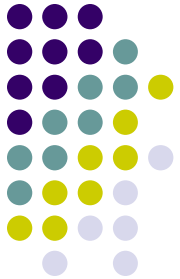
- De Criação
  - Abstraem o processo de criação de instâncias (objetos), oferecendo flexibilidade no que é criado, por quem, como e quando.
- Estruturais
  - Tratam de compor classes e objetos para formar estruturas grandes e complexas
- Comportamentais
  - Padrões comportamentais se preocupam com os algoritmos e a atribuição de responsabilidades entre objetos.

Categorização pelo escopo



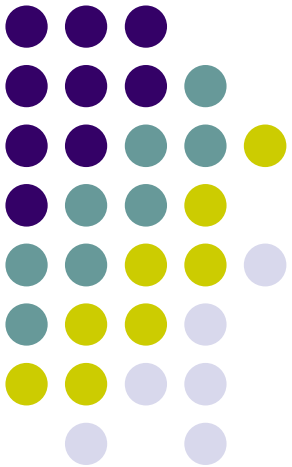
- Classe
  - Lidam com relacionamentos entre classes e subclasses
- Objeto
  - Relacionamentos dinâmicos que mudam na execução

Catálogo GoF

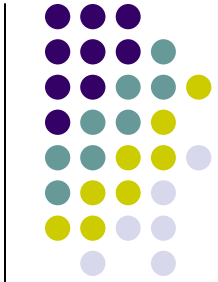


		Propósito		
		Criação	**Estrutural**	**Comportamental**
Escopo	**Classe**	Factory Method	Adapter	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsability Command Iterator Mediator Memento Observer State Stategy Visitor

Observer

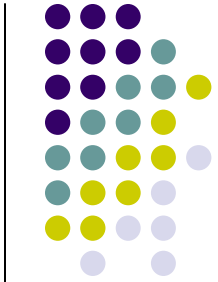


Conteúdo



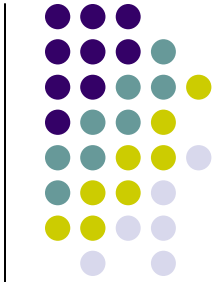
- Introdução
- Objetivo
- Motivação
- Aplicabilidade
- Estrutura
- Participantes
- Colaboração
- Conseqüências
- Implementação
- Exemplos de código
- Aplicações do padrão

Introdução



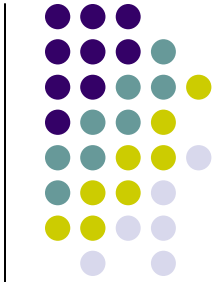
- Desejamos criar uma aplicação que permita alterar/exibir informações contidas em uma única fonte dados através de um gráfico e de uma planilha simultaneamente
- A planilha não deve conhecer o gráfico e vice-versa (isso permite reutilização)
- As modificações podem ser feitas tanto através do gráfico como da planilha

Introdução



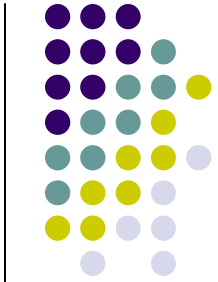
- As alterações feitas em um devem ser refletidas imediatamente no outro
- Poderíamos fazer com que ambos gravassem as alterações em um BD e checassem de tempos em tempos se houve alguma modificação no mesmo
  - Para refletir as modificações rapidamente, é necessário fazer consultas constantemente. Muitas delas seriam desnecessárias

Introdução



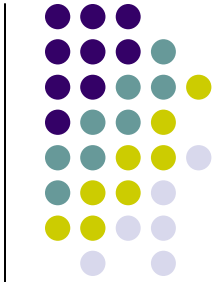
- Essa solução não seria eficiente e não permitiria um grande número de clientes
- E se as informações tivessem que ser armazenadas em um arquivo texto?
  - Seria necessário modificar o gráfico, a planilha e os outros objetos dependentes
- Logo, esse tipo de solução não serve

Objetivo



- Define uma dependência de 1-n (um para muitos) entre objetos a fim de notificar todos os objetos dependentes sobre mudanças ocorridas no objeto
- Também conhecido como
  - Dependentes ou Publicar-Assinar

Motivação

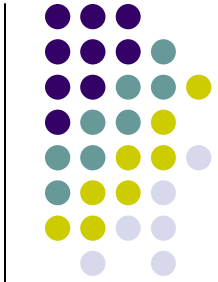


- O particionamento dos sistemas em objetos que cooperam entre si gera a necessidade de manter a consistência entre os objetos relacionados (entre a parte visual e os dados, por exemplo)
- Essa consistência tem que ser feita de uma forma que não crie um acoplamento forte entre as classes

Motivação

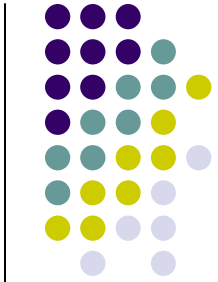
- Os objetos chave do padrão Observer são o **Subject** e o **Observer**
  - Um Subject pode ter um número qualquer de objetos dependentes
  - Todos os Observers são notificados quando o Subject muda de estado
  - Em resposta à notificação, cada Observer vai sincronizar o seu estado com o do Subject

Motivação



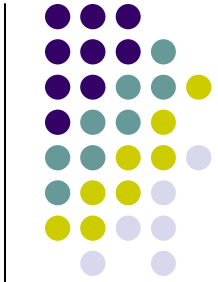
- Este tipo de interação também é conhecida como Publicar-Assinar
  - O Subject é o Editor, ou seja, quem publica as informações sem precisar conhecer a fundo quantos e quem são os seus assinantes
- Em Java, um Subject é conhecido como Source e um Observer como Listener

Aplicabilidade



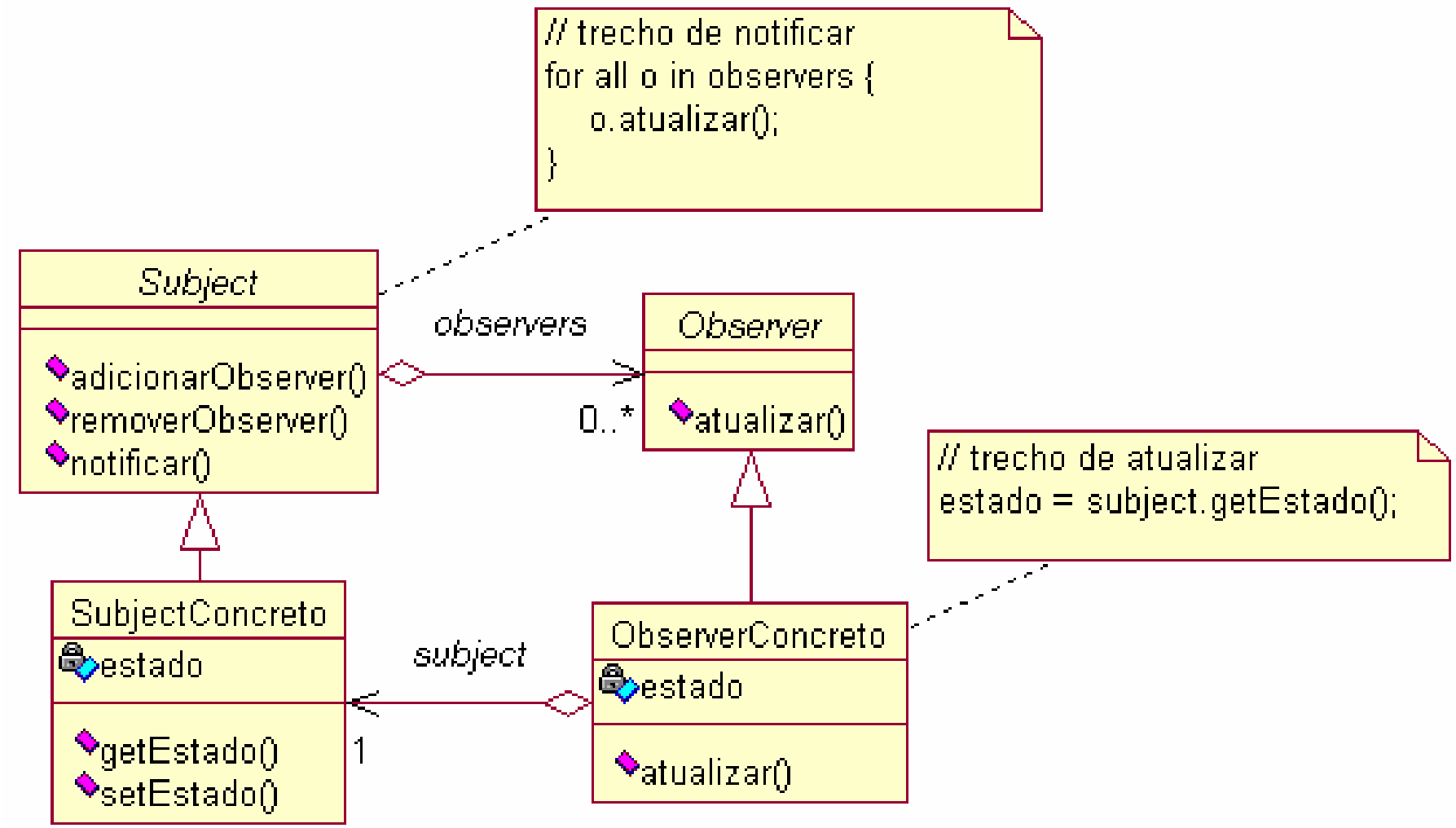
- Usa-se o padrão Observer quando:
  - Uma abstração tem dois aspectos, um dependente do outro
    - Encapsular tais aspectos em objetos separados permite a variação e a reutilização separada
  - Uma mudança em um objeto acarreta mudança em outros e não se conhece antecipadamente quantos objetos precisam ser mudados

Aplicabilidade

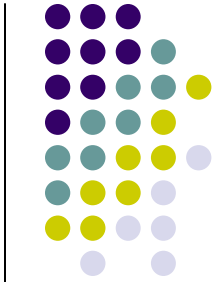


- Um objeto deve ser capaz de notificar outros objetos sem precisar saber exatamente quem são eles, ou seja, quando é necessário que haja um acoplamento fraco entre os objetos

Estrutura

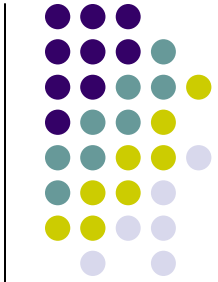

// trecho de notificar
for all o in observers {
o.atualizar();
}
Subject
adicionarObserver()
removerObserver()
notificar()
observers
0..*
Observer
atualizar()
// trecho de atualizar
estado = subject.getEstado();
SubjectConcreto
estado
getEstado()
setEstado()
subject
1
ObserverConcreto
estado
atualizar()

Participantes



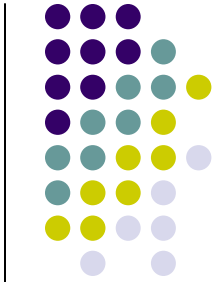
- Subject
  - Conhece seus Observers. Qualquer número de objetos pode observar um Subject
  - Provê uma interface para adicionar e remover Observers
- Observer
  - Define uma interface para objetos que recebem avisos de mudança de estado de um Subject

Participantes



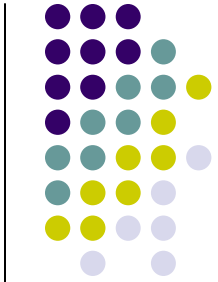
- SubjectConcreto
  - Armazena o estado que interessa aos Observers concretos
  - Envia notificações para os seus Observers quando há alterações no seu estado
- ObserverConcreto
  - Pode manter uma referência para um objeto SubjectConcreto
  - Armazena o estado que deve ser mantido consistente com o do Subject

Participantes



- Implementa a interface de atualização do Observer de forma que mantenha o seu estado consistente com o do Subject

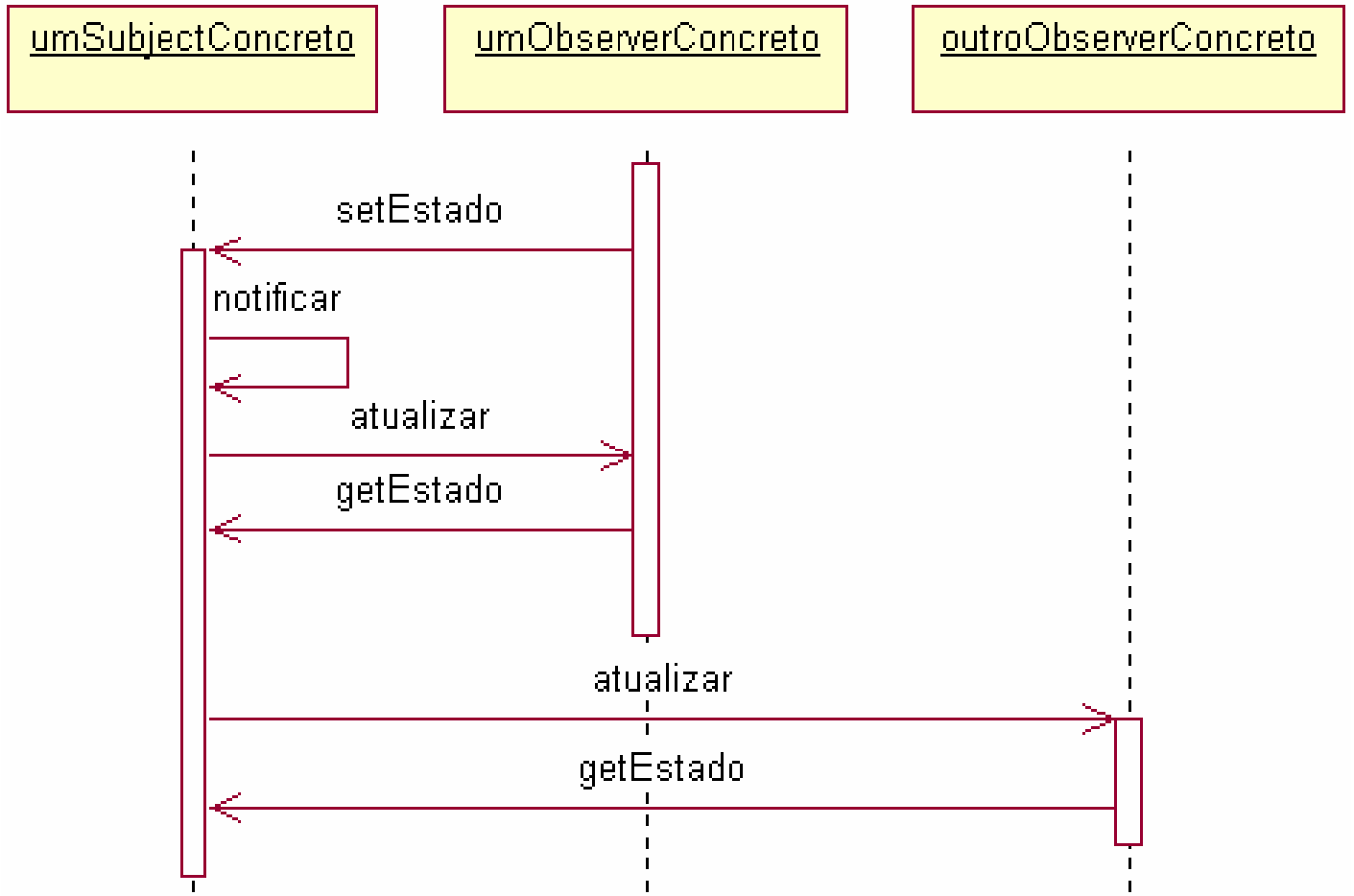
Colaboração



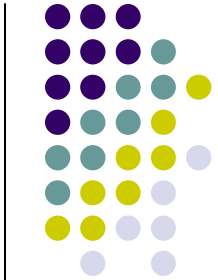
- SubjectConcreto notifica seus observadores sempre que ocorrer uma mudança que possa deixar o estado deles inconsistente
- O ObserverConcreto usa as informações (recebidas durante a notificação ou obtidas junto ao SubjectConcreto quando ocorreu a notificação) para atualizar o seu estado

Colaboração

- Diagrama de seqüência

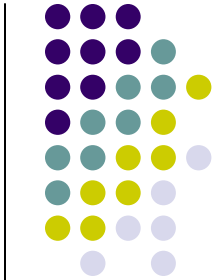


Conseqüências



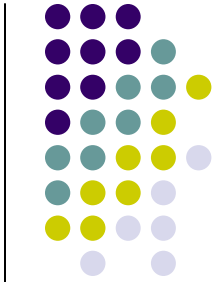
- Suporte a comunicação em broadcast
  - O Subject faz broadcast da notificação. Os Observers têm total liberdade para fazer o que quiserem, inclusive nada
  - Isso permite adicionar e remover Observers dinamicamente

Conseqüências



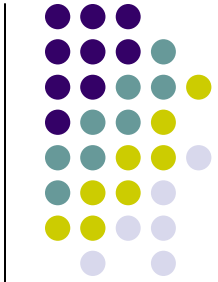
- O acoplamento entre Subject e os Observers é pequeno
  - Os observadores só precisam implementar a interface Observer
  - Os objetos envolvidos podem pertencer a camadas diferentes de software

Conseqüências



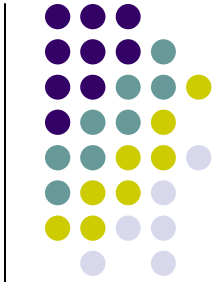
- Permite variar Subjects e Observers independentemente
  - Pode-se utilizar Subjects concretos sem reutilizar seus Observers concretos e vice-versa
  - Pode-se adicionar Observers sem afetar o Subject ou os outros Observers

Implementação



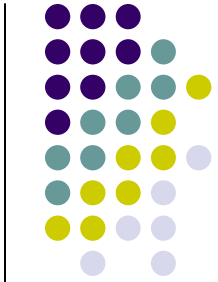
- Como fazer o mapeamento entre Subjects e Observers
  - Armazenando internamente em cada Subject as referências para todos os seus Observers (solução mais cara em termos de memória)
  - Usar uma estrutura de dados associativa (como uma tabela hash) e compartilhá-la entre os Subjects (solução que exige mais processamento)

Implementação



- Observar mais de um Subject simultaneamente
  - É necessário estender a interface para que os Observers identifiquem quem é o Subject
- Quem dispara as notificações
  - O próprio Subject ao mudar de estado
  - O cliente que chama as operações que mudam de estado (não é uma boa pois ele pode esquecer)

Implementação

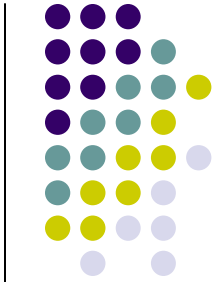


- O modelo **Push** e o **Pull**
  - **Push**: O Subject passa durante a notificação informações detalhadas sobre o seu estado. É menos reutilizável já que expõe detalhes do Subject aos Observers
  - **Pull**: O Subject passa poucas informações sobre o seu estado. Isso exige que os Observers descubram quais foram as modificações, o que pode não ser muito eficiente. Entretanto, é mais reutilizável

Implementação

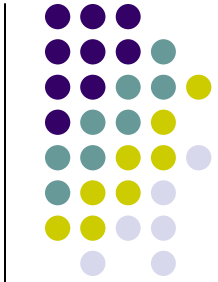
- É necessário que o Subject esteja em um estado consistente antes de iniciar a notificação
  - Os Observers podem precisar obter mais informações sobre o estado do Subject (além das que foram passadas durante a notificação)

Exemplos de código



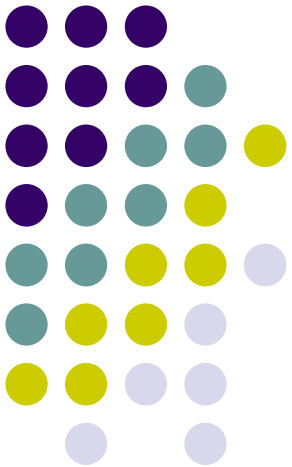
- RelogioEvent.java
- RelogioListener.java
- Relogio.java
- TesteRelogio.java (main)
- RelogioSaidaPadrao.java
- RelogioFrame.java

Observer

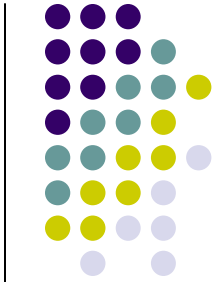


- Aplicações do padrão
  - Em frameworks para criação de interfaces gráficas
  - Em componentes que podem ser conectados dinamicamente
  - Em Java, principalmente nas API's AWT e Swing, e em componentes Java Beans

Factory Method



Introdução

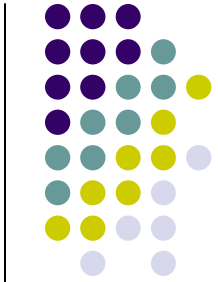


- Consideremos um framework para criação de editores de texto:
  - O framework traz uma classe chamada **Aplicação** que é responsável por gerenciar os documentos
  - **Documento** é uma interface (ou classe abstrata) que deve ser implementada (ou estendida) para cada editor de texto criado

Introdução

- A classe **Aplicação** só exige que os documentos com os quais irá interagir implementem a interface **Documento**
- Cada editor de texto é uma nova aplicação (subclasse de **Aplicação**) e possui um ou mais tipos de documento (classes concretas que implementam a interface **Documento**)
- A classe **Aplicação** sabe “quando” criar um documento mas não sabe “qual” documento (cada editor lida com um documento diferente)

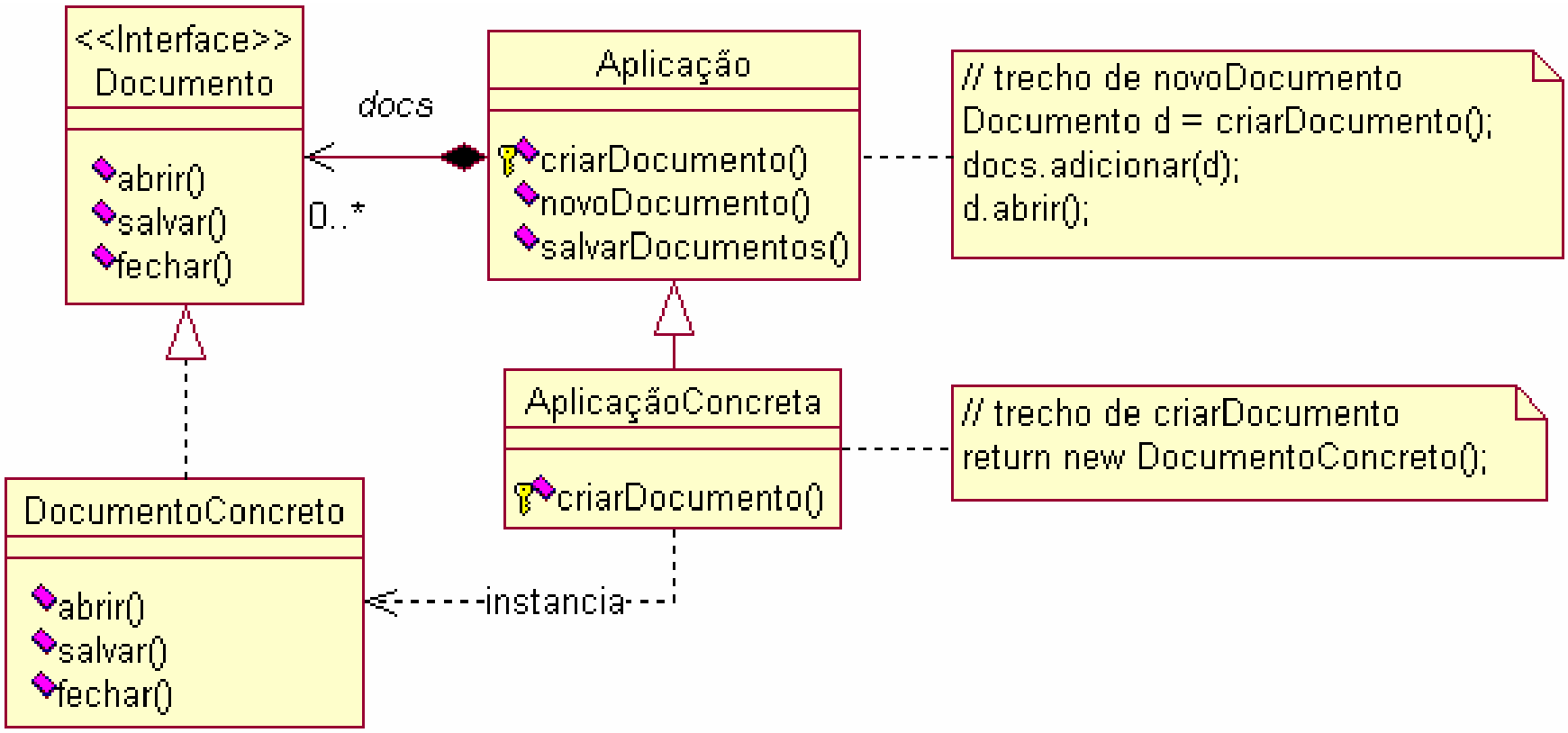
Introdução



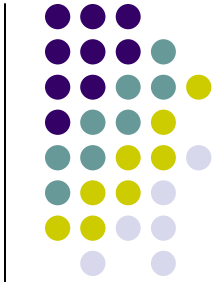
- A classe **Aplicação** deixa que as aplicações concretas (suas subclasses) especifiquem a classe do documento que irão instanciar
- O processo de criação dos documentos é “escondido” em um método, que é implementado pelas subclasses de **Aplicação**.
- O “momento” da criação de um documento continua sendo definido pela classe **Aplicação**

Introdução

- Diagrama UML simplificado do framework



Objetivo

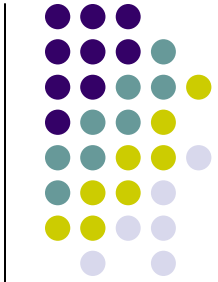


- Definir uma **interface** para criar objetos de forma a deixar as subclasses decidirem qual classe instanciar
- O Factory Method deixa que as subclasses façam a instanciação
- Também conhecido como
  - Construtor virtual

Motivação

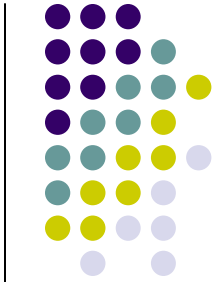
- Frameworks usam classes abstratas para definir e manter relacionamentos entre objetos
- Um framework é freqüentemente responsável por criar esses objetos também
- Um framework deve instanciar classes, mas ele só tem conhecimento das classes abstratas, que não podem ser instanciadas

Motivação



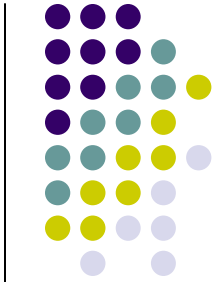
- O Factory Method encapsula o conhecimento de qual é a classe concreta e move esse conhecimento para fora do framework

Aplicabilidade



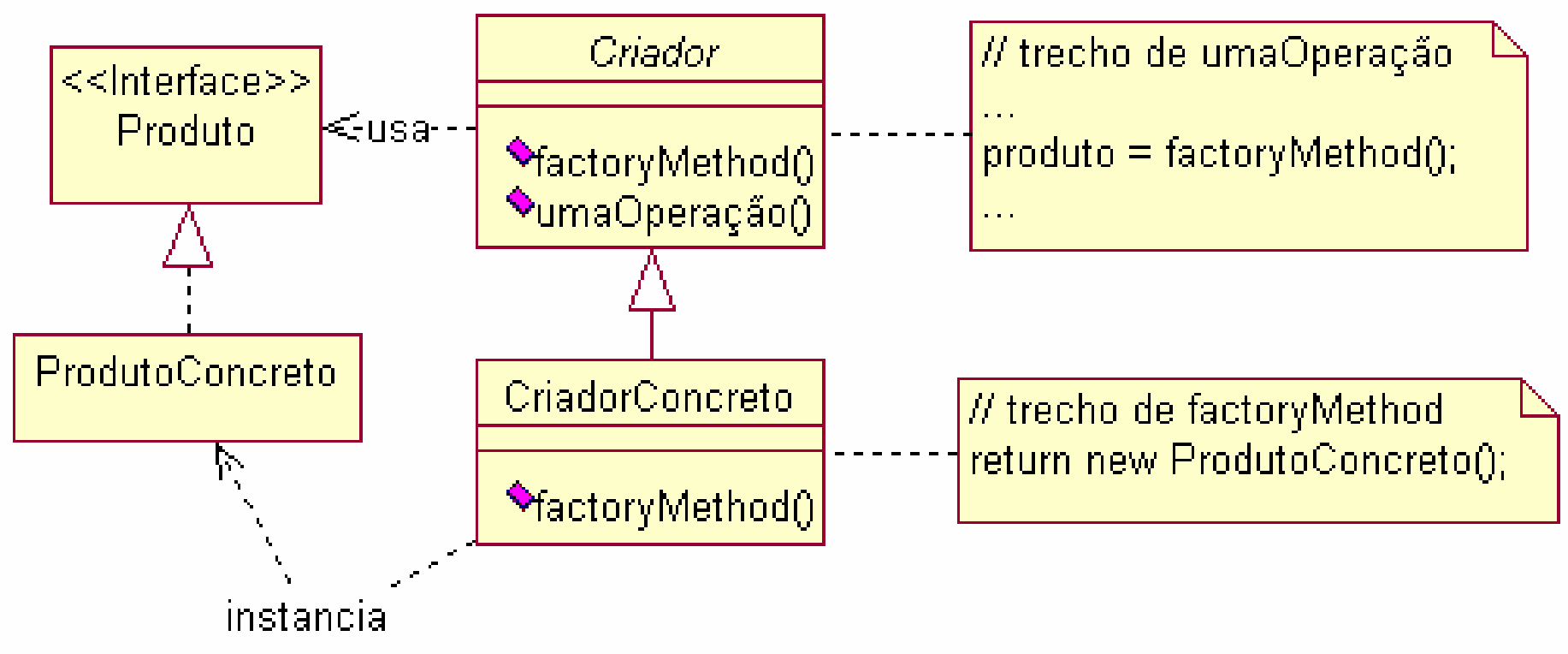
- Pode-se usar o padrão Factory Method quando:
  - uma classe não pode antecipar que classe será instanciada
  - uma classe deseja que as suas subclasses especifiquem a classe que será instanciada

Aplicabilidade

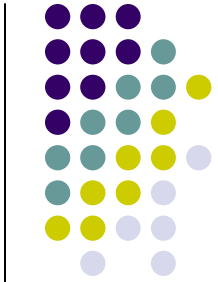


- classes delegam responsabilidades para uma entre várias subclasses de apoio e queremos localizar num ponto único o conhecimento de qual subclasse está sendo usada (diferentes implementações de coleções, por exemplo)

Estrutura

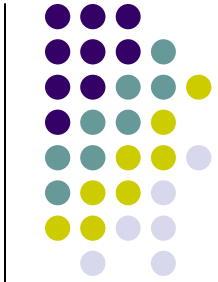

<<Interface>>
Produto
usa
Criador
factoryMethod()
umaOperação()
// trecho de umaOperação
...
produto = factoryMethod();
...
ProdutoConcreto
CriadorConcreto
factoryMethod()
// trecho de factoryMethod
return new ProdutoConcreto();
instancia

Participantes



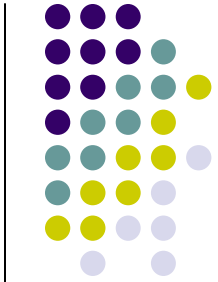
- **Produto**: define a interface dos objetos criados pelo Factory Method
- **ProdutoConcreto**: implementa a interface Produto

Participantes



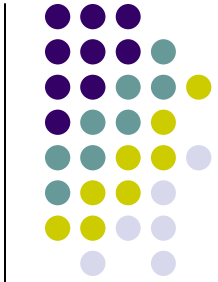
- **Criador**: declara o Factory Method que retorna um objeto do tipo Produto
  - O Criador pode ser uma classe abstrata, uma interface e até uma classe concreta que tenha tem uma implementação padrão para o Factory Method (retorna um objeto com algum tipo ProdutoConcreto padrão)
  - Pode chamar o Factory Method para criar um produto do tipo Produto

Participantes



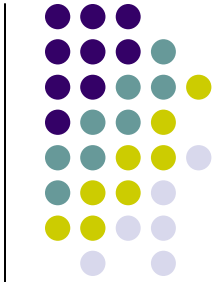
- **CriadorConcreto**: faz a sobreposição (override) do Factory Method para retornar uma instância de ProdutoConcreto

Colaboração



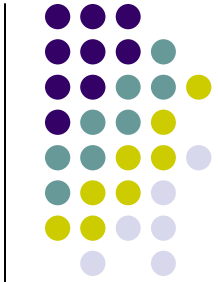
- Criador espera que suas subclasses definam o Factory Method para que ele possa retornar uma instância apropriada do ProdutoConcreto

Conseqüências



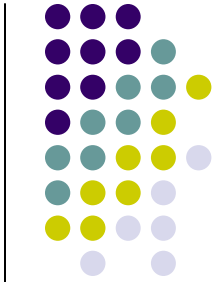
- Factory Methods eliminam a necessidade de colocar classes específicas da aplicação no código
  - O código só lida com a interface Produto
  - Como conseqüência, o código pode lidar também com qualquer classe ProdutoConcreto

Conseqüências



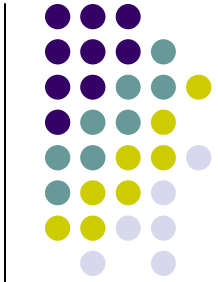
- Provê ganchos para subclasses
  - Criar objetos dentro de uma classe com um Factory Method é sempre mais flexível do que criar objetos diretamente
  - O Factory Method provê um gancho para que subclasses forneçam uma versão estendida de um objeto

Implementação



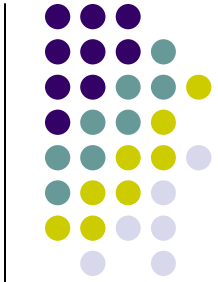
- As duas variantes dos Factory Methods:
  - **Abstratos**: quando não há implementação padrão. A escolha da classe do Produto fica a cargo de uma subclasse do Criador
  - **Com implementação padrão**: quando há uma implementação padrão que faça sentido para Produto e deseja-se permitir que uma subclasse do Criador possa modificá-la

Implementação



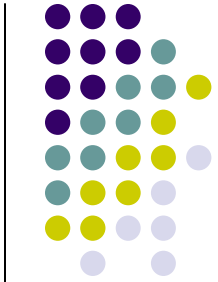
- Deve-se utilizar uma convenção de nomes para destacar os Factory Methods
  - Exemplos: criarX, criaX ou makeX()

Exemplos de código



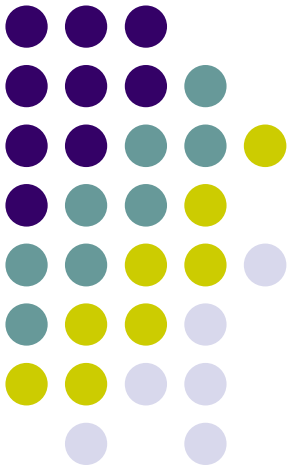
- Log.java
- LogAbstract.java
- LogTela.java
- LogArquivo.java
- TesteLogTela.java (main)
- TesteLogArquivo.java (main)

Aplicações do padrão

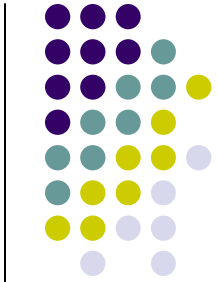


- Frameworks em geral
- Em várias API's de Java, como JDBC

Singleton

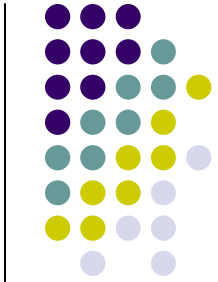


Introdução



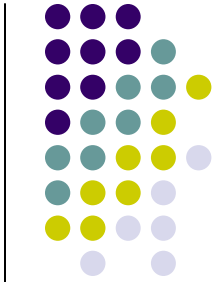
- Consideremos uma classe que permita fazer um log de mensagens
  - Essa classe deve estar disponível em várias partes do sistema para que seja possível gravar mensagens de erro ou de depuração
  - Deverá haver uma única instância dessa classe, para evitar que as mensagens saiam "embaralhadas", como em **singleton.TesteLogTela**

Objetivo



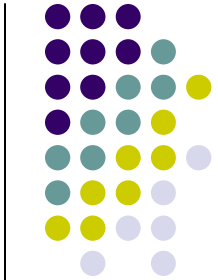
- Certificar que uma classe só possui uma instância
- Oferecer um ponto de acesso único e global

Motivação



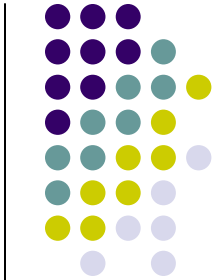
- Às vezes é importante para algumas classes que haja uma única instância (objeto) por aplicação, seja por regras de negócio ou para economizar recursos
- Variáveis globais permitem o acesso global, mas não garantem a existência de um único objeto por classe

Motivação



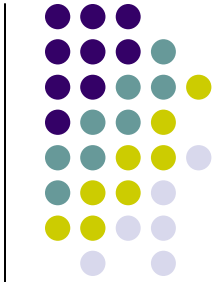
- Exemplos
  - Gerenciador de Janelas
  - Leitor/Gravador de um Arquivo de Configurações ou de Erros de uma aplicação
  - Pool de Conexões a um SGBD
  - Parser para documentos XML
  - Uma conexão de rede

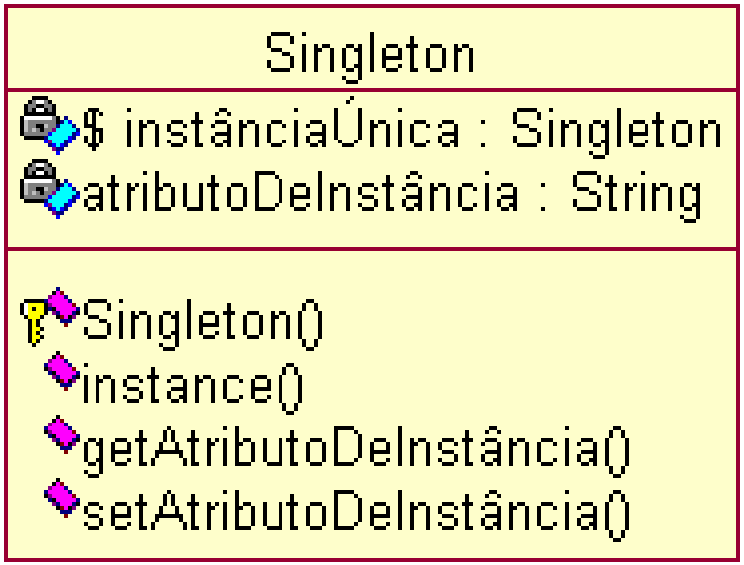
Aplicabilidade



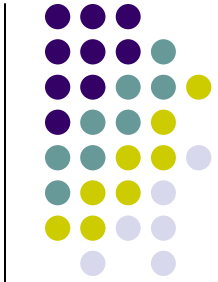
- Quando deva existir uma única instância de uma classe e esta seja deva ser acessível a todos os clientes de um ponto de acesso bem conhecido
- A instância única deveria ser extensível (através da criação de subclasses), e os clientes deveriam ser capazes de usá-la sem modificar seu código

Estrutura



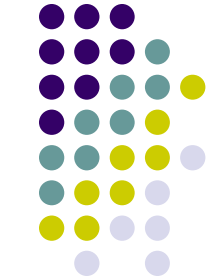

Singleton
$ instânciaÚnica : Singleton
atributoDeInstância : String
Singleton()
instance()
getAtributoDeInstância()
setAtributoDeInstância()

Participantes



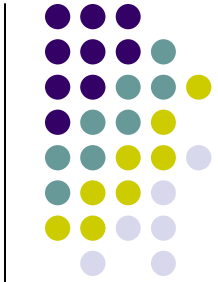
- Singleton
  - Define uma operação de classe “**instance**” que dá aos clientes o acesso à instância única
  - Deve ser responsável por criar a sua instância única

Colaboração



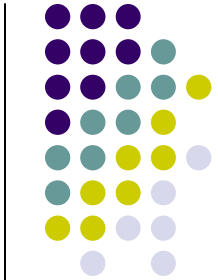
- Os clientes só acessam a instância **Singleton** através da operação **instance**
- Qualquer tentativa de obter uma instância de outra forma deve ser vetada

Conseqüências



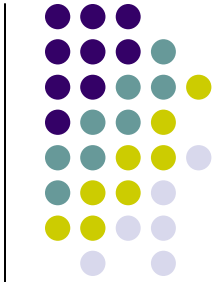
- Benefícios trazidos pelo padrão Singleton
  - Acesso controlado à instância única
    - Como a própria classe encapsula sua única instância, ela pode ter controle sobre como e quando os clientes a acessam
  - Espaço de nomes reduzido
    - Evita a poluição do espaço de nomes com variáveis globais que armazenam instâncias únicas

Conseqüências



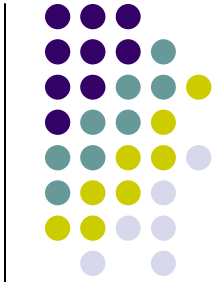
- Permite o refinamento de operações
  - A classe Singleton pode ser estendida e as aplicações podem passar a utilizar a instância da nova classe de forma fácil
- Permite um número variável de instâncias
  - O padrão Singleton permite que se mude de idéia e seja permitida a existência de mais de uma instância para a classe.
    - Apenas a operação que dá acesso à instância única será modificada
    - Pode-se utilizar essa abordagem para controlar o número de instâncias usadas pela aplicação

Conseqüências



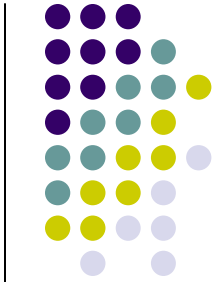
- Mais flexibilidade que operações de classe
  - Uma alternativa ao Singleton é usar atributos e operações de classe. A desvantagem dessa abordagem é que métodos de classe não podem ser estendidos nas subclasses

Implementação



- Garantindo a instância única
  - Os objetos são instanciados através de uma chamada ao seu construtor
  - A forma mais direta de garantir a existência de uma **instância única** é fazer com que essa chamada seja feita **dentro de um método da classe do próprio objeto**

Implementação

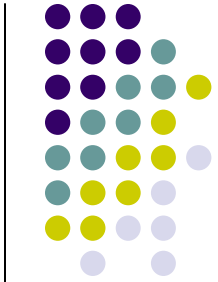


- O método de classe chama o construtor e guarda a instância criada por ele em um atributo de classe
- Chamadas posteriores a esse método retornam a instância criada anteriormente

Implementação

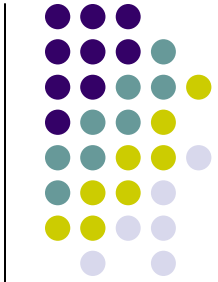
- Protegendo o construtor (public, protected ou private?)
  - A forma mais segura de impedir que um objeto seja instanciado é protegendo o construtor
  - Utilizando o nível de visibilidade **protected**, para permitir que qualquer subclasse possa chamar o construtor da superclasse ou **private** se isso não for desejado
  - Ambas protegem o construtor de chamadas indesejadas

Implementação



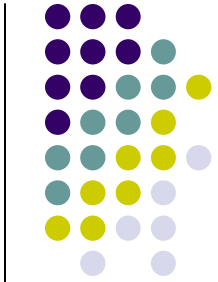
- Utilizando subclasses de forma transparente para o cliente
  - O cliente espera que o método **instance** retorne um objeto da classe Singleton ou de alguma subclasse dela (Princípio da Substituição)
  - A única alteração que deve ser feita é alterar o objeto que é instanciado no método **instance** para um descendente de Singleton. O código cliente fica inalterado

Implementação



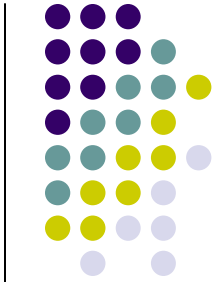
- Tomando cuidado com concorrência
  - A concorrência é um aspecto que deve ser levado a sério ao utilizar o padrão Singleton
  - Quando oferecemos um único objeto a vários clientes corremos o risco de que ocorra acesso concorrente ao objeto, podendo (se não tratado corretamente) ocasionar inconsistências imprevisíveis

Implementação



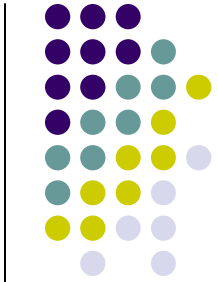
- Em Java, deve-se usar o modificador **synchronized** na assinatura dos métodos ou blocos **synchronized** para controlar o acesso concorrente em:
  - Métodos de instância
    - O mesmo objeto vai ser compartilhado por vários clientes (que podem rodar em Threads independentes)
  - Método de classe **instance**
    - Impedir que dois objetos sejam criados mas somente um seja a verdadeira instância única

Exemplos de código



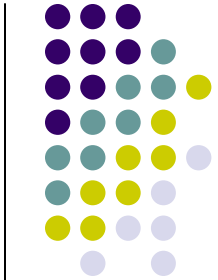
- Texto.java (main)
- LogSingleton.java
- TesteLogTela.java (main)
- TesteLogSingleton.java (main)

Aplicações do padrão



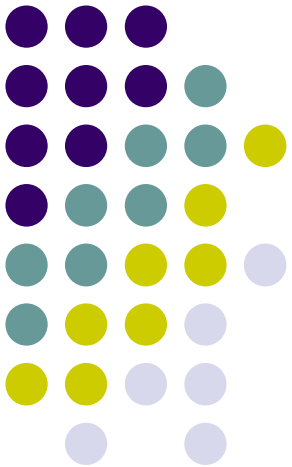
- Um exemplo na linguagem Java é o da classe java.lang.Runtime
  - A classe java.lang.Runtime não possui um construtor e a instância única de Runtime é obtida a partir do método **getRuntime**, equivalente ao **instance**

Aplicações do padrão

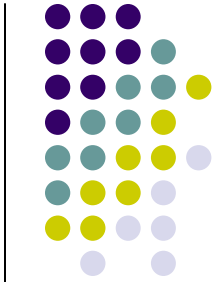


- A classe java.lang.Runtime classe é responsável por encapsular funções de sistema dependentes de plataforma, tais como:
  - Iniciar um processo fora da JVM
  - Encerrar a execução do interpretador
  - Oferecer informações sobre memória
  - Forçar a coleta automática de lixo
  - Carregar bibliotecas dinâmicas

Decorator

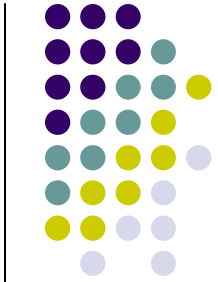


Introdução



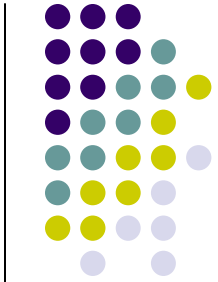
- Desejamos estender uma classe Lista que só permite inserir, pegar e remover objetos para que passe a oferecer funcionalidades como:
  - Sincronização (para controle de concorrência)
  - Suporte a Eventos
  - Garantir que será somente para leitura

Introdução



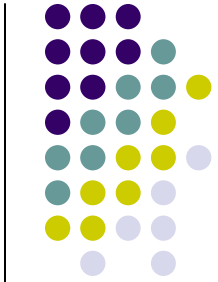
- A primeira idéia é criar subclasses que implementem as funcionalidades desejadas
  - ListaSincronizada
  - ListaComEventos
  - ListaNaoModificavel

Introdução



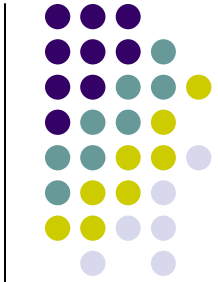
- Mas e se desejarmos mais de uma funcionalidade ao mesmo tempo?
- Nesse caso teremos uma explosão de classes. As novas classes (além das 3 anteriores) poderiam ser chamadas:
  - ListaNaoModificavelSincronizada
  - ListaComEventosNaoModificavel
  - ListaComEventosSincronizada
  - ListaComEventosNaoModificavelSincronizada

Introdução



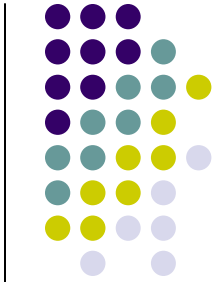
- Essa solução tem alguns problemas:
  - Torna a hierarquia bastante complexa
  - Não é muito flexível
  - O número de classes cresce segundo a fórmula **2^n-1**, onde n é o número de funcionalidades.
    - Isso significa que com apenas 5 funcionalidades teríamos 31 subclasses de Lista!

Objetivo



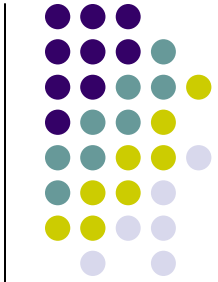
- Adicionar responsabilidades a um objeto **dinamicamente**
- Decoradores oferecem uma alternativa à herança para estender funcionalidade
- Também conhecido como
  - Wrapper

Motivação



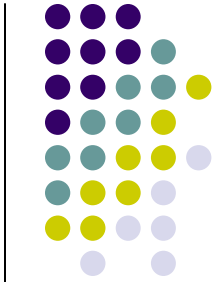
- Apesar de herança ser uma maneira de adicionar responsabilidades, ela exige que a escolha das responsabilidades seja feita estaticamente
- Algumas vezes queremos adicionar responsabilidades a objetos individuais e não a uma classe inteira

Motivação



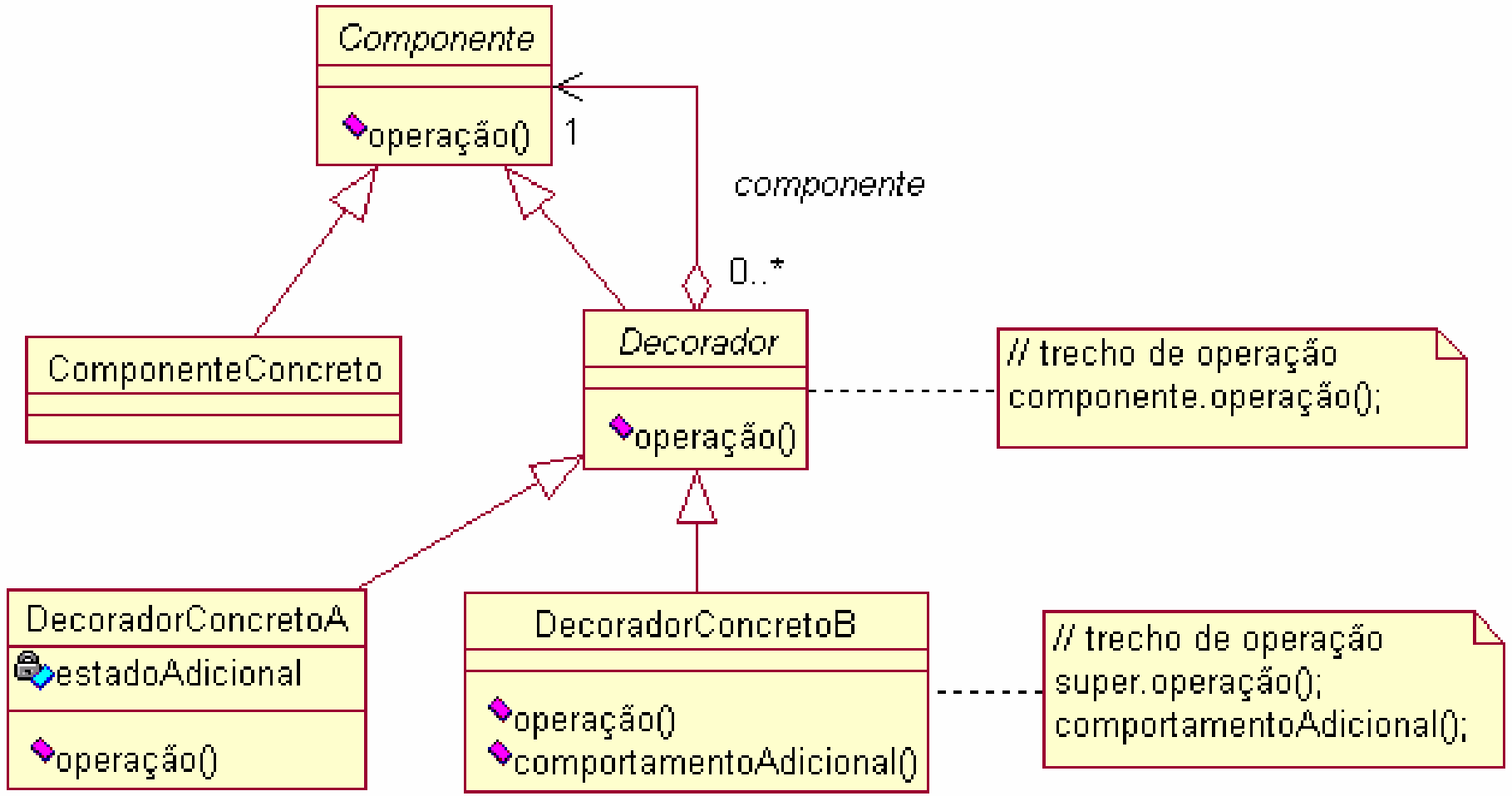
- Às vezes as responsabilidades de um objeto mudam (aumentam ou diminuem) com o tempo e isso requer uma solução flexível

Aplicabilidade

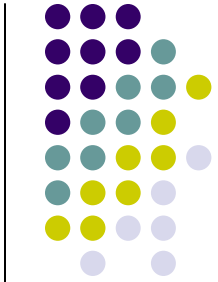


- Deve-se usar o Decorator:
  - Para adicionar responsabilidades a objetos individuais **dinamicamente** e **transparentemente**, isto é, sem afetar outros objetos
  - Quando as responsabilidades adicionadas devem ser facilmente removíveis
  - Quando extensão através de herança é impraticável

Estrutura


Componente
operação()
1
componente
0..*
ComponenteConcreto
Decorador
operação()
// trecho de operação
componente.operação();
DecoradorConcretoA
estadoAdicional
operação()
DecoradorConcretoB
operação()
comportamentoAdicional()
// trecho de operação
super.operação();
comportamentoAdicional();

Participantes

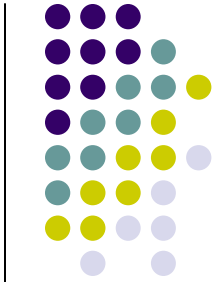


- **Componente**: define a interface
- **ComponenteConcreto**: define uma classe à qual responsabilidades podem ser adicionadas
- **Decorador**: Mantém uma referência a um objeto Componente e define uma interface compatível com a do Componente
- **DecoradorConcreto**: Adiciona responsabilidades ao Componente

Colaboração

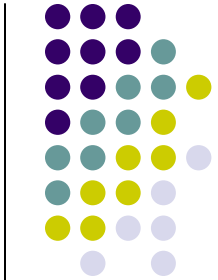
- O Decorador encaminha pedidos ao Componente
- Pode opcionalmente adicionar operações antes ou depois deste encaminhamento

Conseqüências



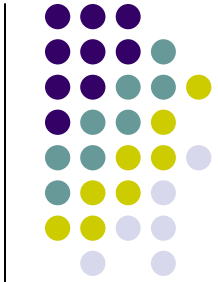
- Mais flexível do que herança estática
  - Pode adicionar e remover responsabilidades dinamicamente
  - Pode até adicionar responsabilidades mais de uma vez (Lista com mais de um tratador de evento)

Conseqüências



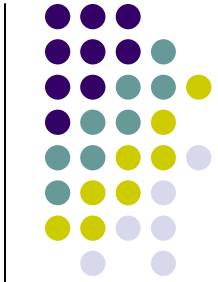
- Evita classes com características demais no topo da hierarquia
  - Ao invés de tentar prover todas as características numa classe complexa e customizável, permite definir classes simples e adicionar funcionalidade de forma incremental com objetos decoradores
  - Desta forma, as aplicações não arcam com os custos das características que não precisam

Conseqüências



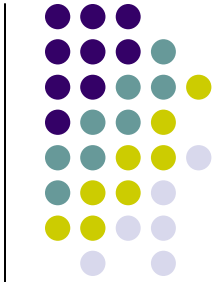
- Um decorador e seu componentes não são idênticos
  - Um decorador age como se fosse uma “capa” sobre o componente
  - Não se pode fazer comparações de identidade pois estaríamos comparando o decorador e não o componente

Conseqüências



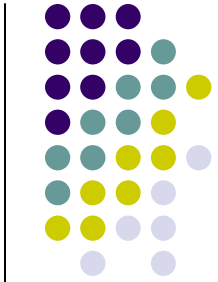
- Gera muitos objetos pequenos
  - Um projeto que usa decoradores freqüentemente resulta em sistemas com muitos objetos pequenos e parecidos
    - Os decoradores são fáceis de serem utilizados desde que sejam bem entendidos
    - Podem ser difíceis de entender e depurar

Implementação



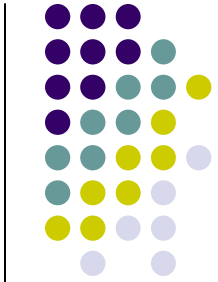
- A interface do decorador deve ser compatível com a do componente que decora
- Em alguns casos não é necessário criar uma classe abstrata para os decoradores, ou seja, o encaminhamento de pedidos para o Componente será feito em cada DecoradorConcreto

Implementação



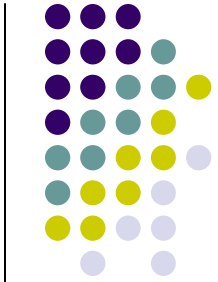
- Os Componentes devem ser mantidos enxutos
  - A classe Componente deve ser mantida enxuta e os dados ser definidos nos Componentes Concretos
  - Caso contrário, os decoradores (que herdam os dados de Componente) ficam "pesados”, tornando inviável a aplicação do padrão

Exemplos de código



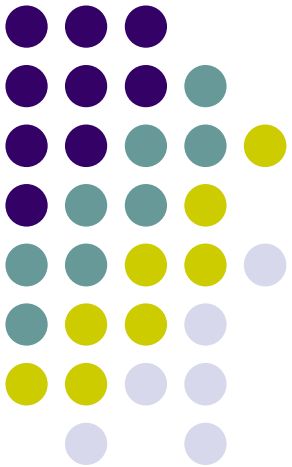
- Exemplos de código
  - Lista.java
  - ListaArray.java
  - ListaNaoModificavel.java
  - ListaSincronizada.java
  - ListaComEventos.java
  - ReceptorEventosLista.java
  - TesteLista.java (main)

Aplicações do padrão

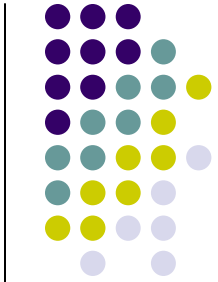


- Aplicações do padrão
  - API Swing de Java para criação de GUI's
  - API de coleções de Java
  - API de entrada/saída de Java
  - Containers para Servlets/JSP
  - Containers para componentes EJB

Iterator

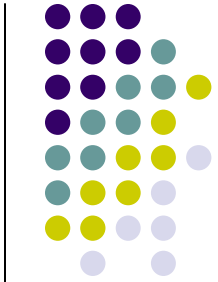


Introdução



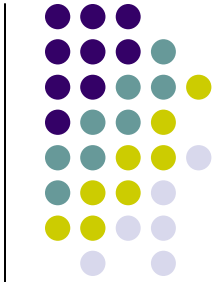
- Desejamos criar uma classe para converter estrutura de dados. Exemplo: copiar o conteúdo de uma fila para uma árvore ou vice-versa
  - Cada estrutura de dados oferece interface e protocolo diferentes para prover acesso aos seus elementos

Introdução



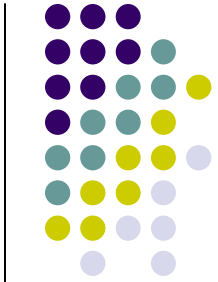
- Algumas estruturas restringem o acesso aos seus elementos, como em Filas e Pilhas
  - Não é possível caminhar por seus elementos sem modificar as estruturas (acessar o elemento do meio de uma pilha sem retirar os que estão sobre ele)
  - Seria necessário ter acesso à implementação para "burlar" o protocolo de acesso
  - Isso quebra um dos pilares da OO: o encapsulamento

Objetivo



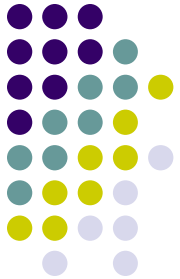
- Prover uma forma de acessar seqüencialmente os elementos de um objeto agregado sem expor sua representação interna
- Também conhecido como
  - Cursor

Motivação



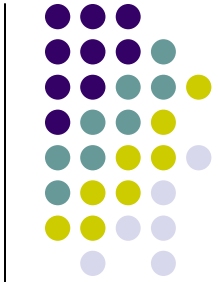
- Deseja-se isolar o uso de uma estrutura de dados da sua representação interna. Isso permite mudar a estrutura sem afetar quem a utiliza
- Às vezes é necessário permitir que mais de um cliente faça o caminhamento simultaneamente

Motivação



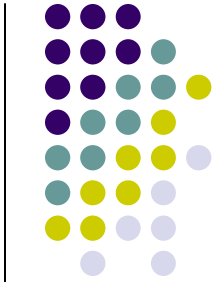
- Para determinadas estruturas, pode haver formas diferentes de caminhamento e queremos encapsular a forma exata de caminhamento. Por exemplo:
  - Uma fila pode ser acessada nos sentidos frente=>fundo ou fundo=>frente
  - Uma pilha pode ser percorrida do topo para a base e da da base para o topo
  - Pode ser necessário criar um “filtro" que só retorne certos elementos

Motivação



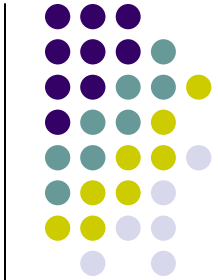
- A idéia do padrão Iterator é retirar da coleção (ou estrutura de dados) a responsabilidade de acessar e caminhar na estrutura, colocando essa responsabilidade num novo objeto separado chamado de iterador
- O iterador é responsável por manter as informações de estado necessárias para saber até onde foi a iteração

Motivação



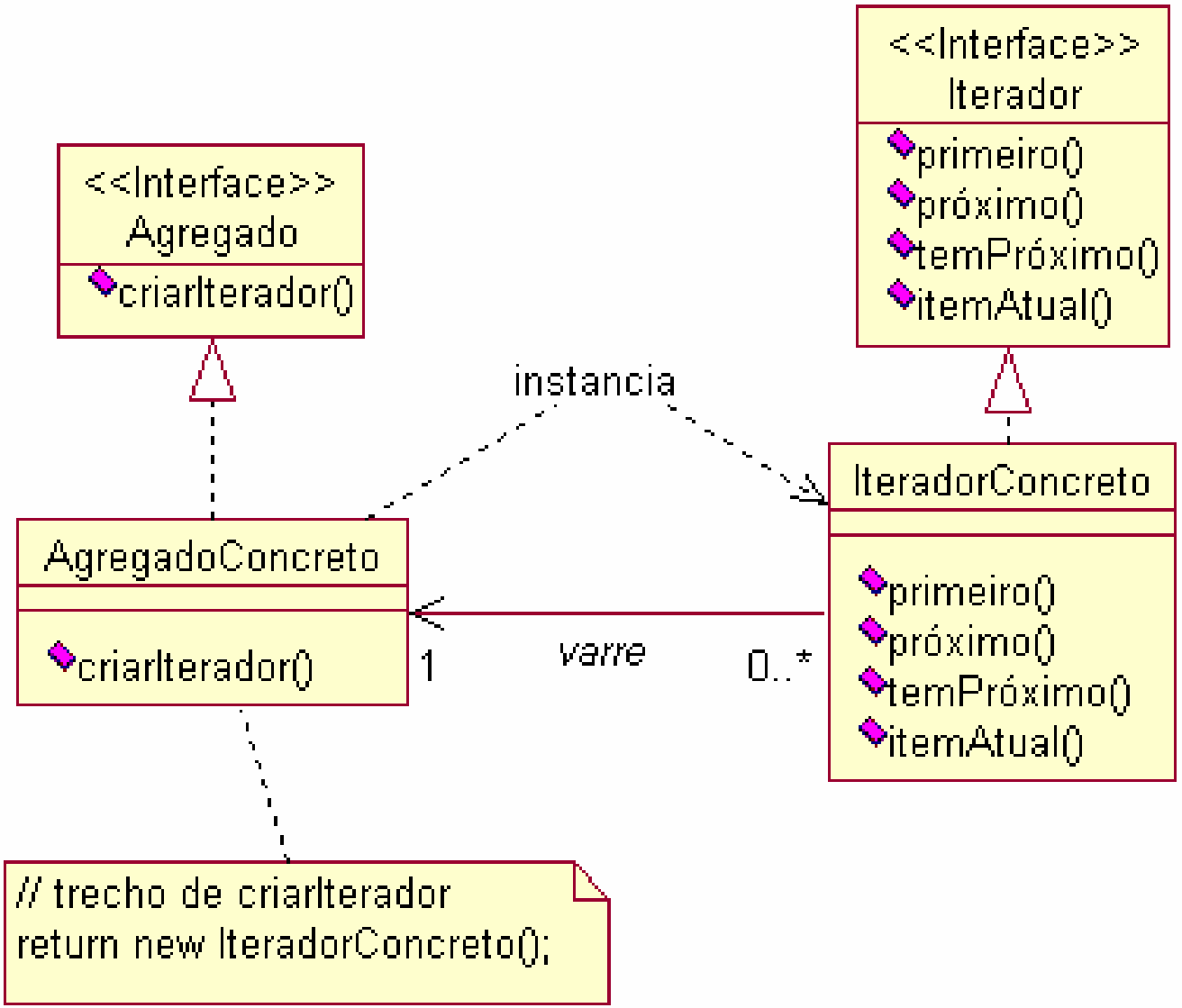
- Um iterador deve ter um interface suficientemente genérica e simples de forma que possa ser usado para varrer todas as coleções
- Criando o iterador:
  - O iterador depende da coleção a ser varrida, sendo necessário que seja criado pela mesma
  - A criação pode ser feita através de um **factory method** na classe da coleção

Aplicabilidade

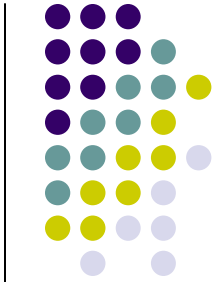


- O padrão Iterator deve ser usado para:
  - Acessar o conteúdo de um objeto agregado sem expor sua representação interna
  - Suportar múltiplas formas de caminhamento
  - Prover uma interface única para varrer estruturas agregadas diferentes

Estrutura

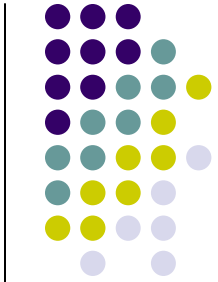

<<Interface>>
Agregado
criarIterador()
<<Interface>>
Iterador
primeiro()
próximo()
temPróximo()
itemAtual()
instancia
AgregadoConcreto
criarIterador()
1
varre
0..*
IteradorConcreto
primeiro()
próximo()
temPróximo()
itemAtual()
// trecho de criarIterador
return new IteradorConcreto();

Participantes



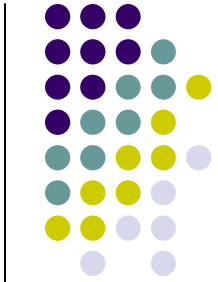
- Iterador
  - Define uma interface para acessar e varrer elementos
- IteradorConcreto
  - Implementa a interface Iterador
  - Mantém a posição corrente (e qualquer outro estado) no caminhamento do agregado

Participantes



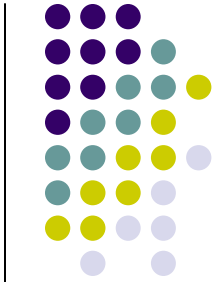
- Agregado
  - Define uma interface para criar um objeto Iterador
- AgregadoConcreto
  - Implementa a interface de criação do Iterador para retornar o IteradorConcreto apropriado

Colaboração



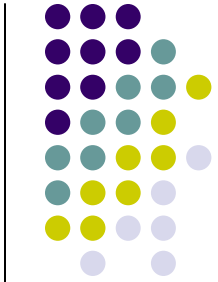
- O IteradorConcreto mantém o objeto corrente no agregado e pode fornecer o objeto consecutivo no caminhamento

Conseqüências



- Simplifica a interface do Agregado
  - As operações para varrer o agregado ficam localizadas no Iterador, permitindo que a interface do Agregado fique mais enxuta
- Mais de um caminhamento pode ficar pendente em um agregado
  - Cada iterador guarda as informações para manter o estado do caminhamento

Conseqüências



- Suporta variações no caminhamento de um agregado
  - Agregados complexos podem ser varridos de diferentes formas:
    - Uma árvore, por exemplo, pode ser varrida “em ordem”, “pós-ordem” ou “pré-ordem”
    - Uma fila pode ser varrida do fundo para a frente e vice-versa

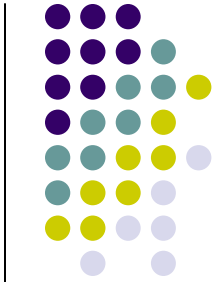
Implementação

- Quem controla a iteração
  - **Iterador interno**: o agregado é quem controla o caminhamento. Para cada elemento, o iterador aplica uma operação definida pelo cliente
  - **Iterador externo**: o cliente é quem controla o caminhamento através das operações oferecidas na interface do Iterador

Implementação

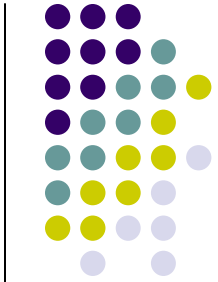
- Um iterador interno só deve ser usado quando um iterador externo é difícil de implementar
  - Para coleções complexas, manter o estado da iteração pode ser difícil e caro (caminhamento em árvores, por exemplo)
- Quem define o algoritmo de caminhamento
  - O agregado: o iterador seria uma espécie de cursor que indicaria a posição corrente
  - O iterador: o iterador, além de guardar o estado do caminhamento, define o algoritmo

Implementação



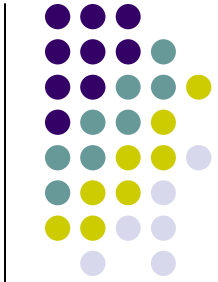
- Iteradores podem ter acesso privilegiado
  - Para permitir uma varredura mais eficiente e, em alguns casos, viabilizar o caminhamento, deve-se expor aos iteradores detalhes de implementação
  - Isso pode ser feito através de:
    - Classes friend em C++
    - Classes internas (Inner Classes) em Java

Implementação



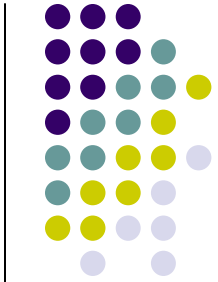
- Operações adicionais
  - Um Iterador pode incluir outras operações como:
    - anterior: volta para o elemento anterior
    - pular: pular um ou mais elementos
    - remover: remove o elemento corrente do agregado

Implementação



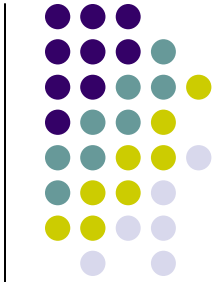
- Quão robusto é o iterador
  - Um iterador robusto permite que modificações no agregado não interfiram nas iterações em andamento. Isso pode ser feito através de:
    - Cópia da coleção na presença de alguma modificação (muito caro em relação a memória)
    - Manutenção de uma lista de modificações sobre a coleção corrente (muito caro em processamento)
  - Um iterador não robusto falha quando uma modificação é feita durante um caminhamento

Exemplos de código



- Iterador.java
- EstruturaDeDados.java
- Fila.java
- Pilha.java
- Processador.java
- TesteIterador.java (main)

Aplicações do padrão



- Aplicações do padrão
  - Comum em linguagens OO. A maioria das bibliotecas de classes oferece iteradores
  - Coleções em Java