

# Programação orientada a objetos: Desenvolvimento avançado em C++

Slide 1

Ivan Luiz Marques Ricarte

DCA/FEEC/UNICAMP

## Objetivos

Slide 2

- Apresentar principais tendências no desenvolvimento de *software*;
- Compreender conceitos da orientação a objetos de modo a obter *software* que pode ser (de fato) reutilizado; e
- Como aplicar esses conceitos para o desenvolvimento de *software* reutilizável em C++.

## Slide 3

**Público-alvo**

Programadores com conhecimento de C++ e com experiência de participação em projetos de sistemas de *software*.



- Como você se descreveria em termos de sua atividade profissional?
- Por que desenvolver *software* é parte de sua atividade?

## Slide 4

**Visão geral**

- Desenvolvimento de *software*
  - Estratégias básicas
  - Tendências atuais
  - Problemas no desenvolvimento de projetos
- Soluções para o desenvolvimento de projetos
  - padrões de projetos
  - técnicas para programação genérica
  - uso efetivo de C++ e STL

Slide 5

## Desenvolvimento de *software*

Slide 6

### Por que se investe tanto no desenvolvimento de *software*?

- Dependência da sociedade em relação aos produtos de *software*
  - Atividades cotidianas
  - Sistemas críticos
- Busca pela melhor qualidade do *software*
  - Confiabilidade
- Uso de *software* no processo de desenvolvimento de *software*

Slide 7

### Evolução do desenvolvimento de *software*

- 1950–60's *Software* orientado pelo *hardware*
- 1960–70's *Software* como produto  
bibliotecas de *software*
- 1970–80's *Software* em sistemas complexos  
popularização de microprocessadores  
sistemas distribuídos
- 1990's–?? *Software* responsável pela maior parte do custo em  
sistemas computacionais

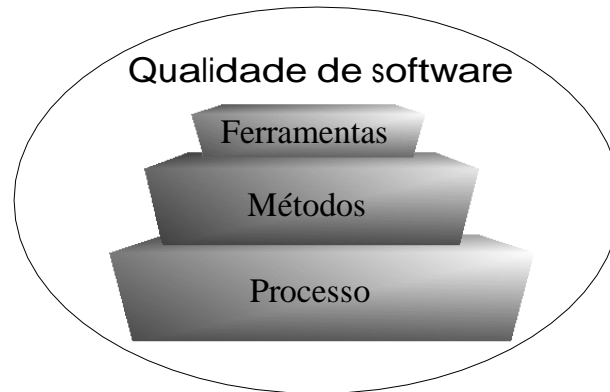
Slide 8

### Disciplina no desenvolvimento de *software*

- O foco na qualidade em desenvolvimento de *software* depende da aplicação consistente e disciplinada de
  - processos:** estabelecimento de uma base sólida para o desenvolvimento de *software*
  - métodos:** estratégias e técnicas para a construção de *software*
  - ferramentas:** suporte automatizado para processos e métodos

## A promessa da Engenharia de *Software*

Slide 9



## Modelos para processos

Slide 10

- Combinações e variações em torno de
  - Análise:** capturar informação sobre o domínio do problema e construir modelos operacionais para o sistema
  - Projeto:** transformar modelos da análise em modelos de elementos computacionais
  - Codificação:** implementar os elementos computacionais do sistema
  - Teste:** encontrar erros na implementação
  - Manutenção:** tudo de novo a cada mudança
- Resultado: código (o produto final)

## Slide 11

**Por que desenvolver *software* é difícil?**

- Frederick Brooks: *No Silver Bullet* para construir *software*
  - Conjunto de construções conceituais
  - Complexo e não-linear
  - Sujeito a mudanças e modificações
  - Invisível e não-visualizável
- Phillip Armour: *software* não é um produto, mas uma forma de armazenar conhecimento
  - desenvolver *software* não é “produzir um produto”, mas adquirir conhecimento

## Slide 12

**Como desenvolver bom *software*?**

- Balanço entre ênfase no produto vs. ênfase no processo
  - construções e linguagens de programação
  - estratégias e metodologias
- “Porém o produto não é o código, mas sim o conhecimento nele embutido”  
*P. Armour*
  - Encerrado o desenvolvimento, todo o *software* deveria ser reescrito

## Slide 13

**As cinco ordens de ignorância**

- OI-0: falta de ignorância
  - conhece alguma coisa e pode demonstrar esse conhecimento
- OI-1: falta de conhecimento
  - não sabe alguma coisa e sabe identificar este fato
- OI-2: falta de consciência
  - não sabe alguma coisa e nem sabe que não sabe
- OI-3: falta de processo
  - OI-2 e não sabe como fazer para descobrir que há coisas que não sabe
- OI-4: meta-ignorância
  - não sabe sobre as cinco ordens de ignorância

## Slide 14

**Ordens de ignorância e desenvolvimento de *software***

- OI-0: sistema funcionando corretamente
  - tem a resposta
- OI-1: variáveis são conhecidas
  - tem a questão
- OI-2: onde muitos projetos começam. . .
  - nem a resposta, nem a questão
- OI-3: onde mora o perigo. . .
  - metodologias de desenvolvimento devem mostrar onde falta conhecimento

## Slide 15

**O papel da orientação a objetos**

- Por si, não é a resposta definitiva às nossas preces
  - Porém, é no momento a melhor maneira de expressar nosso conhecimento sobre *software*
- Diversas metodologias
- Diversas linguagens
  - UML
  - Java
  - ...
  - C++

## Slide 16

**O desenvolvimento orientado a objetos**

- Desenvolvimento não começa na codificação
- Visões da arquitetura do sistema
  - Sistema: coleção de subsistemas
  - Subsistema: agrupamento de elementos
- Modelos
  - Abstração de um sistema semanticamente fechada
- Diagramas sobre aspectos estáticos e dinâmicos
  - Apresentação gráfica de um conjunto de elementos

## A programação orientada a objetos

Slide 17

- Transição dos modelos de projeto para o código é facilitada pelo vocabulário comum
- Linguagens orientadas a objetos permitem expressar diretamente os conceitos usados no desenvolvimento orientado a objetos
  - classes, atributos e métodos
  - objetos
  - associações e composições
  - herança: reaproveitamento de definições

## A velha promessa não cumprida...

Slide 18

- “Com a orientação a objetos, você poderá reaproveitar código já desenvolvido e assim acelerar a produção de *software*.”
- Porém, muitas vezes usamos *software* “genérico” em nossos desenvolvimentos
  - na forma como está ou adaptado às nossas necessidades.

Slide 19

**Como desenvolver *software* que pode ser reaproveitado,  
sem cair nas velhas armadilhas do desenvolvimento de  
*software*?**

Slide 20

**Reaproveitando experiências no desenvolvimento de  
*software***

- Boas experiências
  - Padrões: soluções reconhecidas
- Más experiências
  - Antipadrões: enganos usuais

Slide 21

### Por que estudar antipadrões?

- Cinco em cada seis projetos não são considerados de sucesso
  - Um terço de projetos cancelados
  - Recursos de custo e tempo inadequados
  - Resultados pouco flexíveis ou extensíveis
- Antipadrões: solução para um problema que gera decididamente conseqüências negativas
  - Antipadrões de desenvolvimento: problemas técnicos encontrados pelos programadores
  - Antipadrões de arquitetura: problemas na estrutura do sistema
  - Antipadrões de gerência: problemas na organização de processos e desenvolvimento

Slide 22

### Antipadrões: causas primárias (7 pecados capitais)

**Pressa:** quando o *deadline* se aproxima, qualquer coisa que parece funcionar é aceitável

**Apatia:** não resolver problemas conhecidos

**Mente estreita:** não praticar soluções reconhecidas como efetivas

**Preguiça:** tomar decisões pobres usando a “resposta fácil”

**Avareza:** apegar-se a detalhes excessivos na modelagem

**Ignorância:** não buscar compreender (e.g., migração de código)

**Orgulho:** síndrome do “não-foi-feito-por-nós”

Slide 23

### Como usar antipadrões

- Não é para “caçar bruxas”
- A empresa não necessariamente precisa estar livre de antipadrões
  - endereçar problemas crônicos
- Propósito é desenvolver e implementar estratégias para resolver os problemas decorrentes das más práticas
- Se há problemas, é preciso motivar pessoas a assumirem responsabilidades
  1. Qual é o problema?
  2. O que outras pessoas estão fazendo para contribuir para a solução deste problema?
  3. O que você está fazendo para contribuir para a solução deste problema?

Slide 24

### Antipadrões no desenvolvimento de *software*

- Não basta apontar onde está o problema, mas é preciso indicar caminhos para a solução
- No desenvolvimento de *software*, técnicas básicas de refabricação de programas incluem:
  - Abstração para superclasse
  - Eliminação condicional
  - Abstração agregada

Slide 25

### Abstração para superclasse

- Aplicável a duas ou mais classes similares
  1. Transformar assinaturas de métodos similares em assinaturas comuns
  2. Criar superclasse abstrata
  3. Modificar código para combinar implementações selecionadas
  4. Migrar métodos comuns para superclasse

Slide 26

### Eliminação condicional

- Estrutura e comportamento de uma classe é muito dependente de um comando condicional
  1. Criar novas subclasses correspondentes a cada condição
  2. Migrar o código de ação associado a cada condição para a nova subclasse
  3. Redirecionar as referências às classes para indicar a subclasse adequada
    - Pode afetar construtores, declarações de tipo e invocações a métodos sobrecarregados

Slide 27

## Abstração agregada

- Reorganiza relacionamentos de classes para melhorar estrutura e extensibilidade
- Possíveis formas
  - Transformar relacionamentos de herança em relacionamentos de agregação
  - Migrar classes agregadas para relacionamentos de componentes
  - Migrar relacionamentos de componentes para relacionamentos de agregação

Slide 28

## Antipadrão: A Bolha

*Esta classe é o coração de nossa arquitetura!*

**Forma geral:** Uma classe monopoliza o processamento, outras classes encapsulam dados

- Tipicamente, herança de projeto procedimental (processos vs. dados)

**Sintomas e conseqüências:** classes com grande número de atributos ou métodos, perdendo as vantagens da orientação a objetos e tornando difícil teste e reuso

**Solução:** identificar atributos e operações relacionadas de acordo com contratos coesos, migrando essas coleções de funcionalidades para seus “lares naturais”; revisar associações

Slide 29

### Antipadrão: Fluxo de Lava

*Acho que não é usado, mas não tenho certeza. . . deixe por aí.*

**Forma geral:** fragmentos de código, variáveis de classes aparentemente não relacionados com o sistema

**Sintomas e conseqüências:** segmentos complexos sem documentação, blocos de código comentados sem explicação; se não removido, continua a proliferar pelo sistema e outros desenvolvedores (apressados, intimidados) vão trabalhando ao redor dos fluxos de lava, gerando um sistema impossível de se entender ou documentar

**Solução:** no desenvolvimento, ter uma arquitetura sólida (interfaces estáveis, bem definidas e documentadas) antes de gerar código; na manutenção, trabalho de detetive (descoberta de sistema)

Slide 30

### Antipadrão: Decomposição funcional

*A rotina principal está aqui, na classe Listener.*

**Forma geral:** Desenvolvimento baseado na decomposição funcional, fazendo classes a partir de “subrotinas”

**Sintomas e conseqüências:** classes com nomes de ‘funções’, contendo um único método, e nenhum uso de princípios básicos da orientação a objetos; nenhuma esperança de reusar *software*

**Solução:** definir modelos de análise e de projeto para tentar compreender e explicar o sistema; para classes “fora” do modelo de projeto, tentar combinar com classes existentes ou transformá-las em funções de uso geral

Slide 31

### Antipadrão: Poltergeists

*Eu não sei bem o que essa classe faz, mas certamente é importante.*

**Forma geral:** Classes com ciclo de vida breve, que aparecem brevemente e depois desaparecem

**Sintomas e conseqüências:** Objetos e classes temporários, com associações transientes, levando a modelos de objetos desnecessariamente complexos

**Soluções:** ações associadas a poltergeists devem ser movidas para as classes que elas referenciavam, removendo as “classes fantasmas” do modelo

Slide 32

### Antipadrão: Martelo Dourado

*Quando a única ferramenta disponível é um martelo, todo o resto vira prego.*

**Forma geral:** Todas as soluções de uma equipe usam um produto no qual a equipe tornou-se proficiente

**Sintomas e conseqüências:** Mesmas ferramentas e produtos usadas em produtos conceitualmente diversos, com a arquitetura do sistema sendo melhor descrita pelo produto, ambiente ou ferramenta; resultado em geral podem ter baixo desempenho e escalabilidade, sendo dependentes do vendedor ou da tecnologia

**Solução:** Suportar filosofia de buscar novas tecnologias; projetar e desenvolver sistemas com limites claros para a substituição de componentes individuais.

Slide 33

### Antipadrão: Código espaguete

*Você sabia que essa linguagem suporta mais de uma função?*

**Forma geral:** Programas ou sistemas com pouca estrutura de *software*

**Sintomas e conseqüências:** Métodos muito orientados a processos, com o fluxo de execução ditado pela implementação de objetos; muitos métodos sem parâmetros, usando variáveis de classe (“globais”), sendo de difícil reuso

**Solução:** prevenção (uso apropriado de orientação a objetos); manutenção para limpeza de código (estratégias de refabricação de programas), principalmente quando for acrescentar alguma nova funcionalidade ao código espaguete

Slide 34

### Antipadrão: Programação *Cut-and-Paste*

*Isto que é eficiência: 100000 linhas de código em duas semanas!*

**Forma geral:** Presença de vários segmentos similares de código espalhados pelo sistema

**Sintomas e conseqüências:** Os mesmos *bugs* reaparecendo, apesar de várias correções locais; maior tempo de revisão e inspeção de código; maior custo na manutenção do *software*

**Solução:** Enfatizar estratégia de reuso caixa-preta no desenvolvimento ou re-estruturação do código

Slide 35

### Antipadrões de arquitetura de *software*

**Sistemas encanamento:** integração ponto-a-ponto

**Travamento ao vendedor:** não se esqueça de renovar a licença

- Ingresso do lobo: suportamos padrão X (mas interfaces são proprietárias)

**Arquitetura por implicação:** já fizemos sistemas assim antes

**Projeto por comitê:** *camelo (s.m.)*: cavalo projetado por um comitê

- Canivete suíço: tudo que pensamos foi incluído no projeto

**Reinventar a roda:** nosso problema é diferente dos outros

Slide 36

### Antipadrões de gerência de projetos de *software*

**Paralisia da análise:** é melhor repensar esses modelos de análise para torná-los mais orientados a objetos

**Morte por planejamento:** não podemos começar enquanto não houver um plano completo de programação

**Espigas de milho:** sujeitinho difícil de trabalhar. . .

**Gerenciamento irracional:** as prioridades do projeto são as minhas!

**Falta de gerenciamento:** o que aconteceu de errado? Estava tudo indo tão bem

Slide 37

## Padrões de projeto

Onde estão os caminhos para as boas soluções?

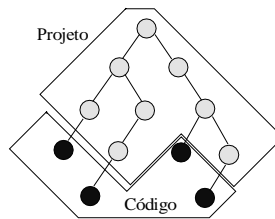
Slide 38

## O que é um padrão de projeto?

- Soluções para problemas específicos em projeto de *software* orientado a objetos
  - Desenvolvidas através da revisão e evolução ao longo de vários projetos
- Descrição geral de um padrão composta por
  - Nome:** criação de um vocabulário
  - Problema:** quando aplicar o padrão
  - Solução:** descrição abstrata de elementos que compõem o projeto
  - Conseqüências:** resultados e compromissos associados à aplicação do padrão

## Por que reusar projetos ao invés de código?

- Pode ser aplicado em mais contextos
  - mais compartilhável
- Ocorre mais cedo no processo de desenvolvimento
  - maior impacto



Slide 39

## Problemas de projeto abordados por padrões

**Identificação de objetos:** auxiliam a identificar abstrações recorrentes de forma genérica

**Granularidade de objetos:** indicam como representar subsistemas como objetos ou suportar vários objetos pequenos

**Especificação de interfaces:** indicam os conceitos chaves que devem (ou não devem) estar na interface de um objeto, assim como relacionamentos entre interfaces

**Especificação de implementação:** indicam que classes devem ser abstratas (puras) ou concretas, embora a implementação deva sempre favorecer referências a interfaces

Slide 40

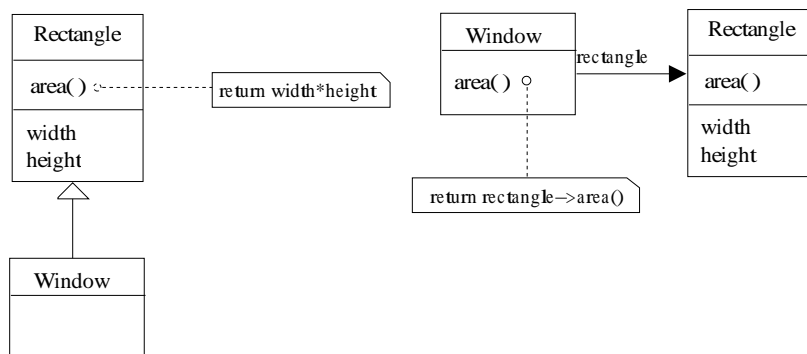
Slide 41

## Padrões, orientação a objetos e reuso

- Objetos, interfaces, classes e herança não garantem reuso
- Abordagens de reuso na orientação a objetos:
  - Herança de classes:** reuso caixa-branca, definição estática (tempo de compilação), simples, mas expõe superclasse
  - Composição de objetos:** reuso caixa-preta, definição dinâmica (obter referência durante execução), mais complexo de compreender (uso de delegação, principalmente)
- Padrões de projeto favorecem equilíbrio desses mecanismos
- Outra abordagem de reuso, não ligada à OO: *templates* (C++)

Slide 42

## Derivação vs delegação



Slide 43

### **Agregação vs associação**

- Agregação: objeto é composto por outros objetos ou um objeto é parte de outro objeto
  - mesmo tempo de vida
- Associação: objeto referencia outro objeto
  - acoplamento menor
- Na programação, construções similares
  - Referências ou ponteiros para objetos da outra classe

Slide 44

### **Potenciais problemas no projeto de sistemas modificáveis: Criar objetos especificando explicitamente sua classe**

- Assume compromisso com uma implementação em particular, podendo comprometer futuras modificações
- Padrões de projeto associados: Método Fábrica, Fábrica Abstrata, Protótipo

## Padrão: Método Fábrica

**Objetivo:** Definir uma interface para criar um objeto, mas deixar que as subclasses decidam qual classe instanciar

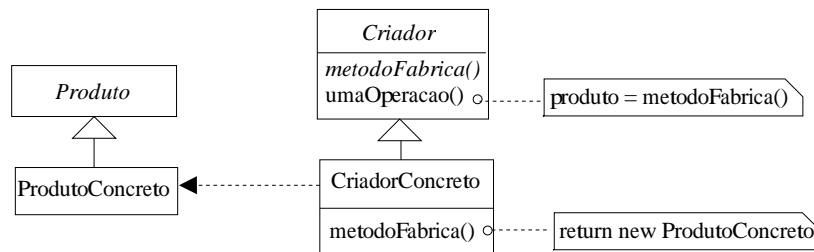
**Motivação:** o sistema sabe que será preciso criar um objeto, mas naquele ponto do código não sabe que tipo de objeto será criado

**Conseqüências:** isola classes específicas da aplicação do código do sistema; oferece um ponto de extensão (*hook*) para subclasses

**Aspectos de implementação:** método fábrica pode ter implementação padrão ou ser abstrato em Criador; pode ter parâmetros para indicar tipo de objeto a criar; em C++, *templates* podem ser utilizados; um padrão de nomeação deve ser utilizado

Slide 45

### Estrutura:



Slide 46

## Padrão: Fábrica Abstrata

**Objetivo:** Oferecer uma interface para criar famílias de objetos relacionados ou dependentes sem especificar suas classes concretas

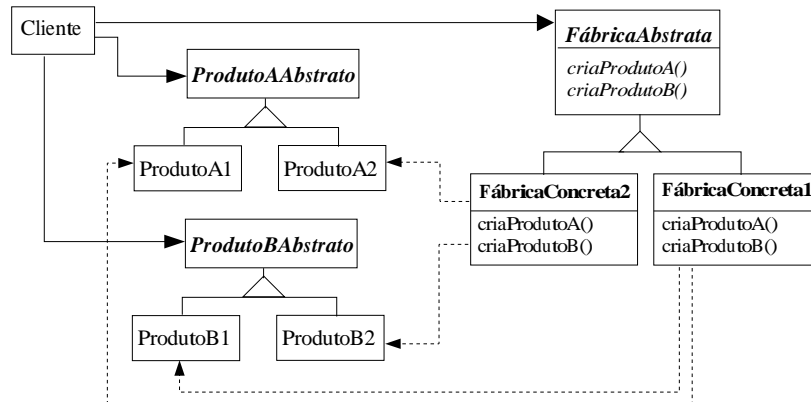
**Motivação:** trabalhar com uma interface no código que seja comum para distintas alternativas de implementação daquele conjunto de funcionalidades

**Conseqüências:** isola classes concretas do sistema; permite facilmente trocar famílias de produtos; promove consistência entre produtos; não é simples estender a fábrica para criar novos tipos de produtos

**Aspectos de implementação:** tipicamente, apenas um objeto do tipo fábrica para uma família de produtos existe no sistema; a criação do produto dá-se tipicamente através de um método fábrica

Slide 47

### Estrutura:



Slide 48

## Padrão: Protótipo

**Objetivo:** Especificar o tipo de objeto que deve ser criado usando uma instância de protótipo e criar novos objetos copiando este protótipo

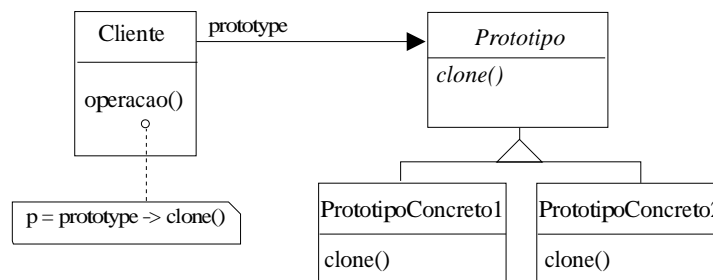
**Aplicabilidade:** sistema deve ser independente de como objetos devem ser criados, compostos ou representados, e

- classes a serem instanciadas são conhecidas apenas no momento da execução, ou
- para evitar construir uma hierarquia de fábricas paralela à hierarquia de classes de produtos

**Conseqüências:** permite acrescentar e remover produtos em tempo de execução; reduz número de subclasses; classes devem implementar um método clone (nem sempre simples)

Slide 49

### Estrutura:



Slide 50

Slide 51

### Potenciais problemas no projeto de sistemas modificáveis: Estar dependente de operações específicas

- Assume compromisso com uma forma de atender a uma requisição; seria melhor não ter essa definição amarrada ao código
- Padrões de projeto associados: Cadeia de responsabilidade, Comando

Slide 52

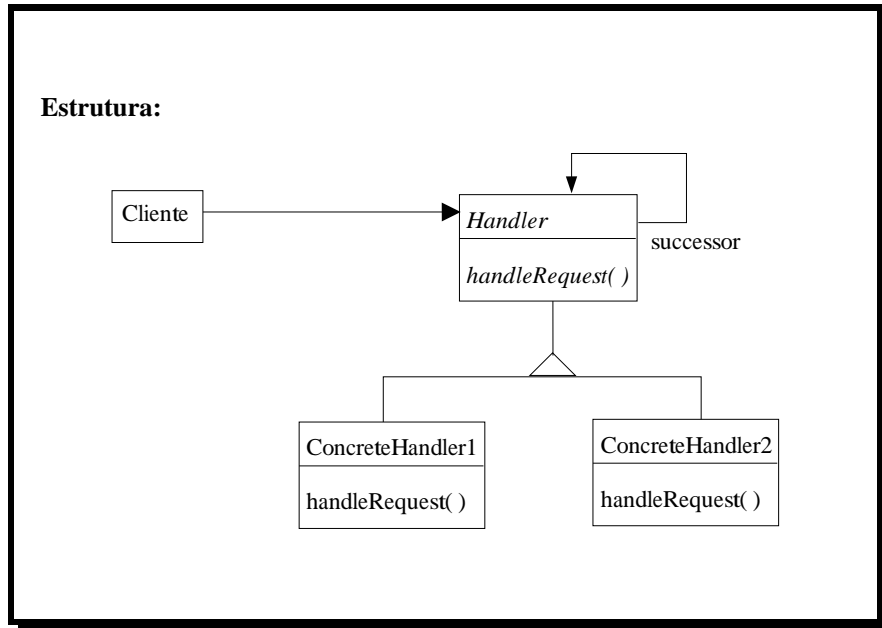
### Padrão: Cadeia de responsabilidade

**Objetivo:** Evitar o acoplamento direto de um solicitante em relação ao atendente de uma requisição dando a oportunidade de ter mais de um objeto respondendo à solicitação. Os atendentes são encadeados e a requisição passada por eles até que um dos objetos a atenda.

**Conseqüências:** reduz acoplamento e obtém maior flexibilidade na atribuição de responsabilidades a objetos, porém não há garantia de que algum objeto atenderá a solicitação.

**Aspectos de implementação:** como implementar a cadeia de sucessores (referências novas ou existentes), como representar as solicitações (métodos, objetos)

Slide 53



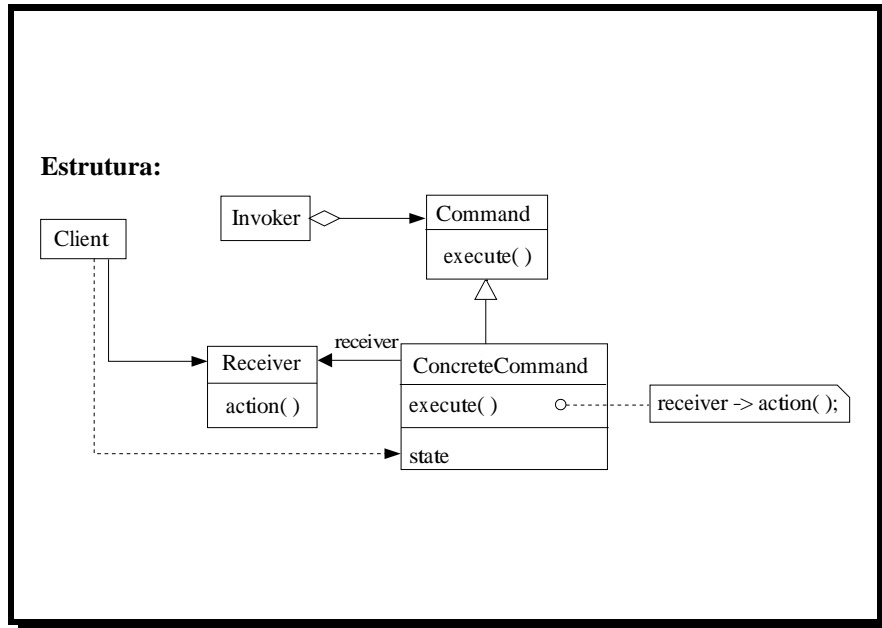
Slide 54

### Padrão: Comando

**Objetivo:** encapsular uma requisição como um objeto, permitindo parametrizar clientes com diferentes solicitações, enfileirar ou registrar solicitações e suportar operações que podem ser desfeitas.

**Conseqüências:** desacopla objeto que invoca o serviço daquele que sabe como executá-lo; solicitações, sendo objetos, podem ser manipuladas e compostas como tais.

Slide 55



Slide 56

**Potenciais problemas no projeto de sistemas modificáveis: Dependem da plataforma de *hardware* e *software***

- Usar diretamente APIs e interfaces para sistemas externos que dependem da plataforma de execução torna portabilidade e mesmo atualização na própria plataforma difíceis
- Padrões de projeto associados: Fábrica abstrata, Ponte

## Padrão: Ponte

Slide 57

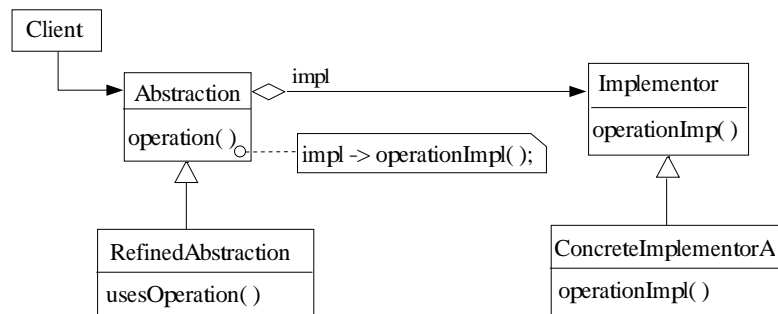
**Objetivo:** desacoplar uma abstração de sua implementação de forma que os dois possam variar independentemente.

**Motivação:** uma forma de evitar o acoplamento definitivo entre abstração e implementação que se dá através de herança

**Conseqüências:** desacopla interface e implementação, melhora extensibilidade e esconde detalhes de implementação de clientes.

Slide 58

**Estrutura:**



Slide 59

### Potenciais problemas no projeto de sistemas modificáveis: Dependem de representações ou implementações de objetos

- Clientes que sabem como um objeto é representado, armazenado, localizado ou implementado podem ter que sofrer modificações quando o objeto muda.
- Padrões de projeto associados: Fábrica abstrata, Memento, Proxy

Slide 60

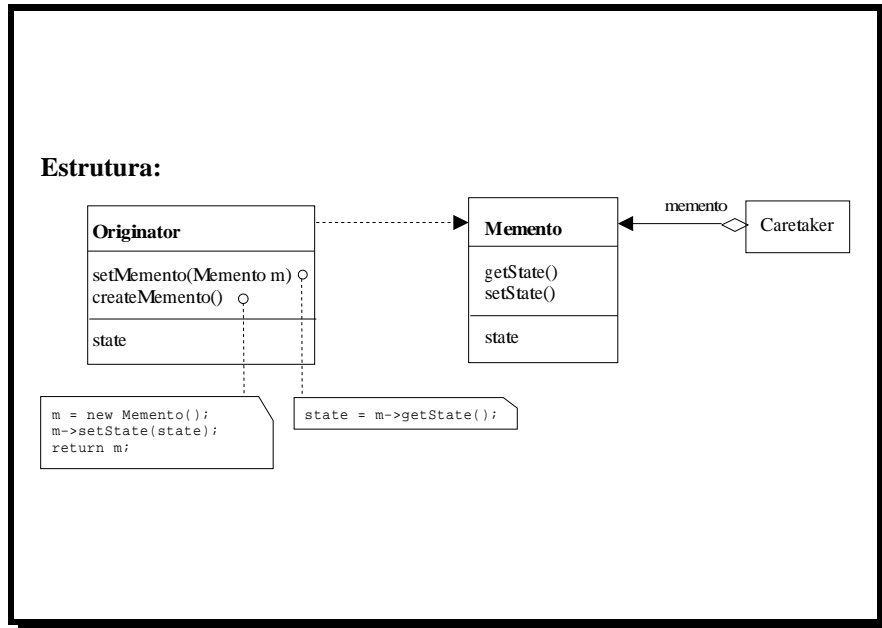
### Padrão: Memento

**Objetivo:** Sem violar encapsulação, capturar e externalizar o estado interno de um objeto de forma que ele possa ser restaurado para esse estado em um momento posterior.

**Motivação:** Um memento é um objeto que armazena o estado interno de um outro objeto (o originador do memento).

**Aplicabilidade:** usar quando um instantâneo (total ou parcial) de um objeto deve ser salvo para posterior recuperação e uma interface direta para obter esse estado exporia detalhes de implementação, quebrando a encapsulação do objeto.

Slide 61



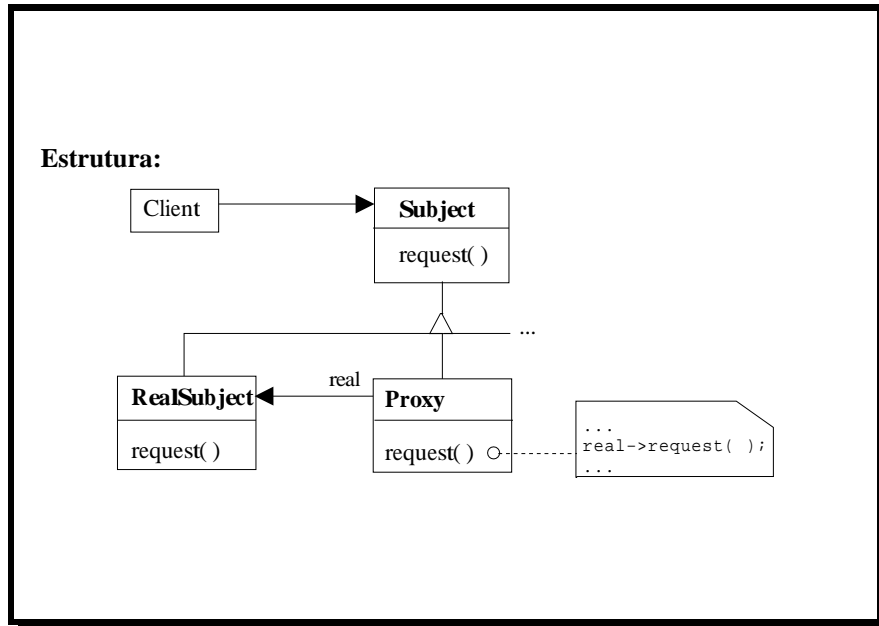
Slide 62

## Padrão: Proxy

**Objetivo:** Oferecer um *surrogate* para outro objeto para controlar o acesso a ele.

**Aplicabilidade:** Sempre que for necessário ter uma referência para um objeto que seja mais versátil ou sofisticada que um simples ponteiro — proxy remoto (referências fora do espaço de endereçamento local), proxy virtual (atrasa criação de objetos “caros” até que haja demanda real), proxy de proteção (controla acesso ao objeto original) e referências “espertas” com funcionalidades adicionais (contar número de referências para liberação automática, carregar objetos persistentes na primeira referência, *locking*).

Slide 63



Slide 64

### Potenciais problemas no projeto de sistemas modificáveis: Dependência de algoritmos

- Algoritmos podem ser estendidos, otimizados ou substituídos durante desenvolvimento ou reuso; se objeto depender de um algoritmo especificamente, também deverá ser alterado nesses casos
- Padrões de projeto associados: Estratégia, Método Gabarito, Iterador

## Padrão: Estratégia

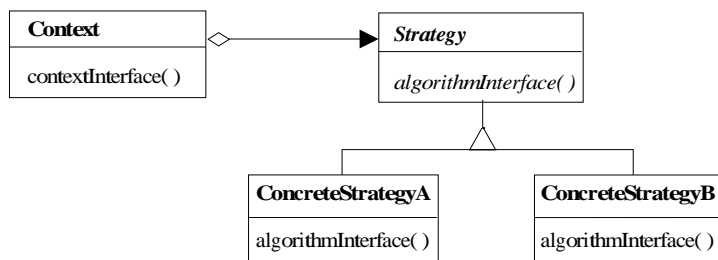
**Objetivo:** definir uma família de algoritmos, encapsulá-los e torná-los intercambiáveis, permitindo que o algoritmo varie independentemente de seus clientes

**Aplicabilidade:** usar quando classes relacionadas diferem apenas em seu comportamento; quando diferentes variantes de um algoritmo são necessárias; quando se deseja encapsular estruturas de dados complexas, específicas do algoritmo; quando a classe define diversos comportamentos com múltiplas ocorrências de um padrão condicional

**Conseqüências:** em famílias de algoritmos relacionados, herança pode fatorar funcionalidades comuns; uma alternativa para derivação direta; cliente exposto às diferentes estratégias disponíveis

Slide 65

### Estrutura:



Slide 66

## Padrão: Método Gabarito

**Objetivo:** definir o esqueleto de um algoritmo em uma operação, postergando alguns passos para subclasses

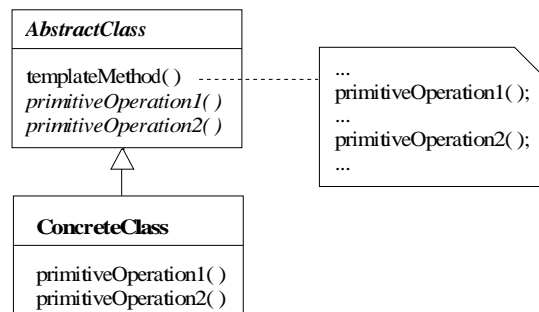
**Motivação:** permite a descrição do algoritmo em termos de operações abstratas, que deverão ser redefinidas nas subclasses

**Aplicabilidade:** usar para implementar as partes invariantes de um algoritmo uma única vez; quando comportamento comum pode ser fatorado em uma superclasse

**Conseqüências:** algumas operações usadas pelo método gabarito podem ser *hooks* (podem ser redefinidas) ou operações abstratas (tem que ser redefinidas); leva ao Princípio de Hollywood (superclasse invoca métodos de classe derivada e não ao contrário)

Slide 67

### Estrutura:



Slide 68

## Padrão: Iterador

Slide 69

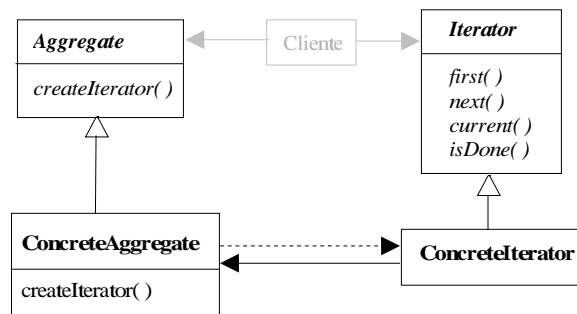
**Objetivo:** oferecer uma forma de acessar os elementos de um objeto agregado seqüencialmente sem expor sua representação interna

**Motivação:** suportar formas (eventualmente, alternativas) de varrer agregados sem ter de incorporar essas funcionalidades à interface do agregado; ter mecanismo uniforme de varrer estruturas agregadas distintas

**Conseqüências:** simplifica a interface do agregado; pode ter mais de uma varredura sobre o mesmo agregado em um dado momento

Slide 70

**Estrutura:**



Slide 71

### Potenciais problemas no projeto de sistemas modificáveis: Acoplamento forte

- Classes fortemente acopladas são difíceis de reutilizar isoladamente, levando a sistemas monolíticos e de difícil manutenção
- Padrões de projeto associados: Fábrica Abstrata, Ponte, Cadeia de Responsabilidade, Comando, Fachada, Mediador, Observador

Slide 72

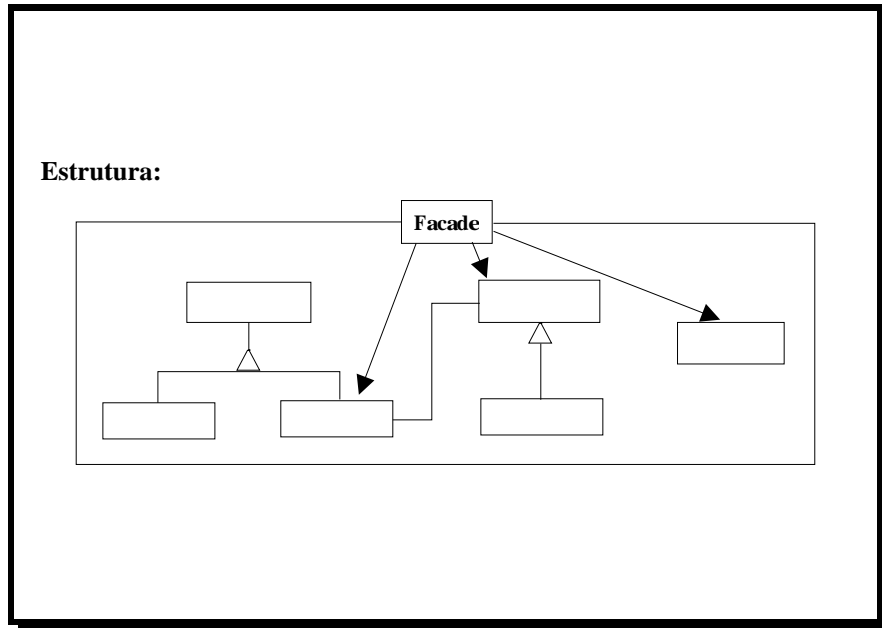
### Padrão: Fachada

**Objetivo:** oferecer uma interface unificada para um conjunto de interfaces em um subsistema

**Motivação:** definir uma interface de nível mais alto para tornar o subsistema mais fácil de utilizar

**Aplicabilidade:** usar quando quiser oferecer uma interface simples para um subsistema complexo; quando houver muitas dependências entre clientes e as classes de implementação de uma abstração; quando quiser estruturar os subsistemas em camadas

Slide 73



Slide 74

### Padrão: Mediador

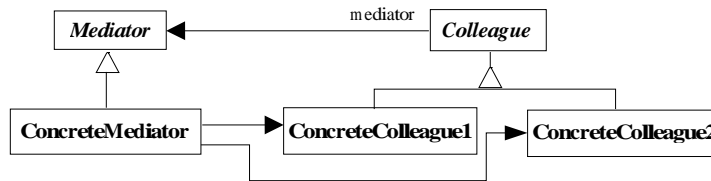
**Objetivo:** definir um objeto que encapsula como conjuntos de outros objetos interagem

**Motivação:** reduzir o número de interconexões entre objetos fazendo com que eles se comuniquem através do mediador

**Conseqüências:** desacopla objetos “colegas”; simplifica protocolos entre objetos; porém, mediador centraliza controle, tornando-se eventualmente um elemento complexo e de difícil reuso

Slide 75

Estrutura:



Slide 76

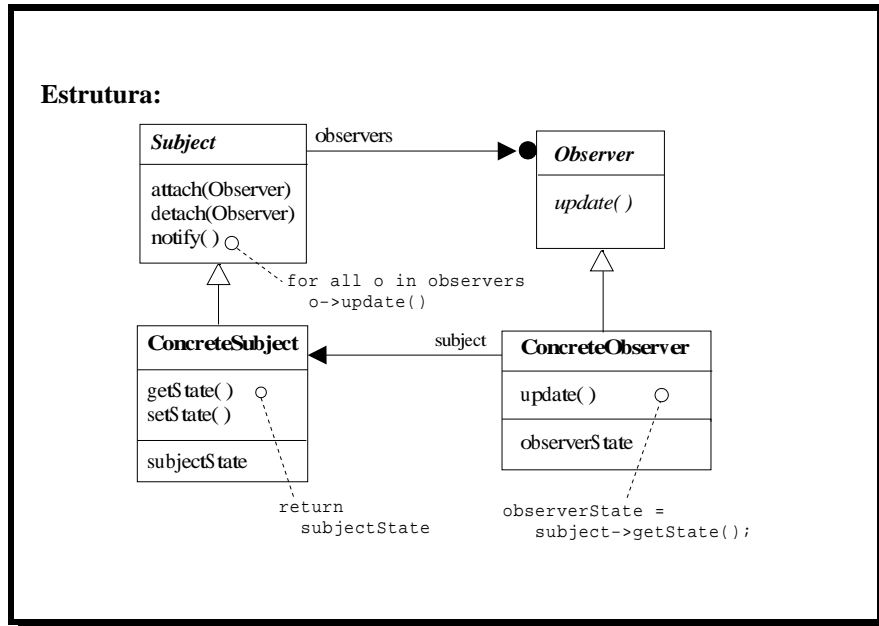
### Padrão: Observador

**Objetivo:** definir uma dependência de um objeto para muitos de forma que, quando um objeto muda de estado, todos os seus dependentes são notificados e automaticamente atualizados

**Motivação:** manter consistência entre objetos da aplicação sem recorrer a um forte acoplamento entre eles

**Aplicabilidade:** usar quando uma abstração tem dois aspectos, um dependente do outro; quando mudança em um objeto requer mudanças em outros; quando um objeto deve poder notificar outros sem assumir nenhum conhecimento sobre quais são esses outros objetos

Slide 77



Slide 78

## Outros padrões de projeto em GoF

**Adaptador:** converte a interface de uma classe em outra interface do tipo que o cliente espera (padrão estrutural);

**Composto:** representa hierarquias parte-todo e permite tratar a composição e os objetos individuais de forma uniforme (padrão estrutural);

**Construtor:** separa a construção de um objeto complexo de sua representação de forma que o mesmo processo de construção possa criar representações diferentes (padrão de criação);

**Decorator:** acrescenta funcionalidades adicionais a um objeto de forma dinâmica (padrão estrutural);

**Estado:** permite que um objeto modifique seu comportamento quando seu estado interno muda (padrão de comportamento);

Slide 79

**Interpretador:** define a representação para uma gramática e um interpretador para sentenças nessa linguagem (padrão de comportamento);

**Peso-pena:** usa compartilhamento para lidar com grande número de pequenos objetos de forma eficiente (padrão estrutural);

**Visitante:** representa uma operação a ser executada nos elementos da estrutura de um objeto;

**Singleton:** garante que uma classe tem apenas uma instância e oferece um ponto de acesso para essa instância (padrão de criação);

## Padrões e *frameworks*

Slide 80

**Padrões de projeto:** descrições de soluções de projeto recorrentes que foram aprovadas pelo uso ao longo do tempo;

**Frameworks:** projeto reutilizável do todo ou de parte de um sistema que é representada por um conjunto de classes abstratas e pela forma que suas instâncias interagem

- menos abstratos que padrões
- tipicamente contêm vários padrões

Slide 81

## *Frameworks e reuso*

- Reuso de projeto
  - um *framework* define um esqueleto de aplicação que pode ser adaptado por um desenvolvedor de aplicação
  - é um tipo de arquitetura voltada para um domínio
- Reuso de código
  - *frameworks* são expressos em linguagens de programação — são programas
  - facilita uso de componentes que se conformem às interfaces do *framework*
  - tornam-se dependentes das linguagens

Slide 82

## *Características de frameworks*

- Modularidade:** detalhes de implementação são encapsulados por trás de interfaces estáveis
- Reusabilidade:** definem componentes genéricos associados às interfaces estáveis que podem ser reutilizados para criar novas aplicações
- Extensibilidade:** definem pontos de adaptação e extensão nas interfaces estáveis (pontos variáveis, métodos *hook*)
- Inversão de controle:** *framework* define seqüência de invocação da aplicação
- Princípio de Hollywood

## Formas de adaptação em *frameworks*

Formas básicas de associar os métodos da aplicação aos métodos do *framework*:

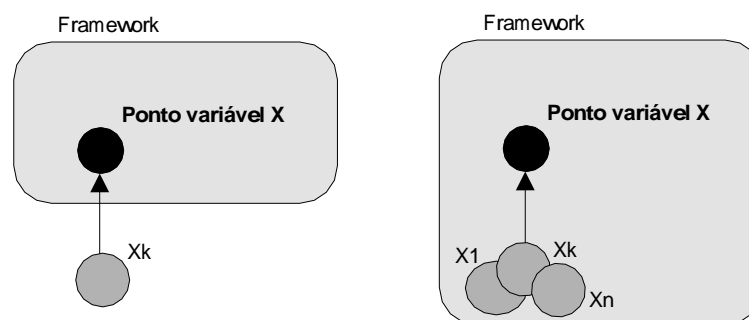
**Adaptação caixa-branca:** reuso por herança — aplicação deve definir classes que estendem as classes abstratas do *framework* e redefinir métodos

**Adaptação caixa-preta:** reuso por composição — aplicação escolhe subclasse concreta (dentre as disponíveis) e utiliza suas funcionalidades via sua interface

**Adaptação caixa-cinza:** oferece alternativas de implementação (como caixa-preta) mas permite implementações específicas (como caixa-branca)

Slide 83

## Pontos de adaptação no *framework*



Slide 84

## Tipos de frameworks

Slide 85

**Frameworks caixa-branca:** todos os pontos variáveis são caixa-branca;

**Frameworks caixa-preta:** todos os seus pontos variáveis são caixa-preta;

**Frameworks caixa-cinza:** apresenta pontos-variáveis caixa-cinza.

## Aspectos de desenvolvimento e utilização dos diferentes tipos de *frameworks*

Slide 86

- *Frameworks* caixa-branca são mais simples de se projetar e desenvolver
  - não precisa oferecer implementações dos pontos variáveis
- *Frameworks* caixa-preta são de utilização mais simples
- *Frameworks* caixa-cinza tendem a caixa-preta
  - Implementações realizadas passam a fazer parte do conjunto de implementações disponíveis

Slide 87

### Desenvolvimento de *software* centrado em *frameworks*

- Três fases principais:
  1. Desenvolvimento do *framework*
  2. Uso do *framework*
  3. Manutenção do *framework*

Slide 88

### Tarefas para o desenvolvimento de *frameworks*

1. identificar domínio específico de aplicação do *framework*
2. determinar os principais casos de uso suportados e atores interagindo com o *framework*
3. determinar padrões/soluções para auxiliar desenvolvimento do *framework*
4. projetar interfaces e componentes do *framework*; mapear atores e papéis para as interfaces
5. desenvolver implementação padrão para interfaces do *framework*
6. descrever e documentar os pontos de extensão do *framework*
7. criar planos e casos de teste

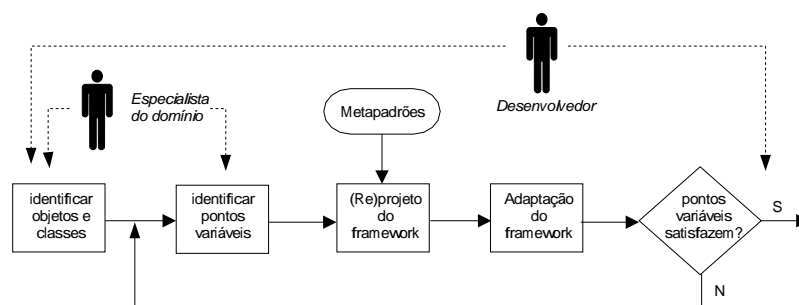
## Desenvolvimento de *framework* baseado em pontos variáveis

Slide 89

- Análise do domínio identifica pontos variáveis
  - quais aspectos do *framework* diferem entre aplicações?
  - qual o grau de flexibilidade desejado?
  - o comportamento flexível precisa ser alterado durante o funcionamento da aplicação?

## Ciclo de desenvolvimento baseado em pontos variáveis

Slide 90



Slide 91

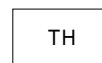
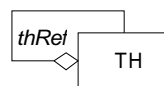
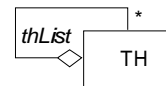
## Metapadrões

- Estabelecem padrões de relação entre classes genéricas (*template classes*) e classes componentes (*hook classes*)
  - Classes genéricas possuem os métodos gabaritos, que definem comportamento abstrato, fluxo de controle genérico, relação entre objetos
  - Classes componentes possuem os métodos componentes, que fazem parte das implementações dos métodos gabaritos e podem ser abstratos, regulares ou novos gabaritos.

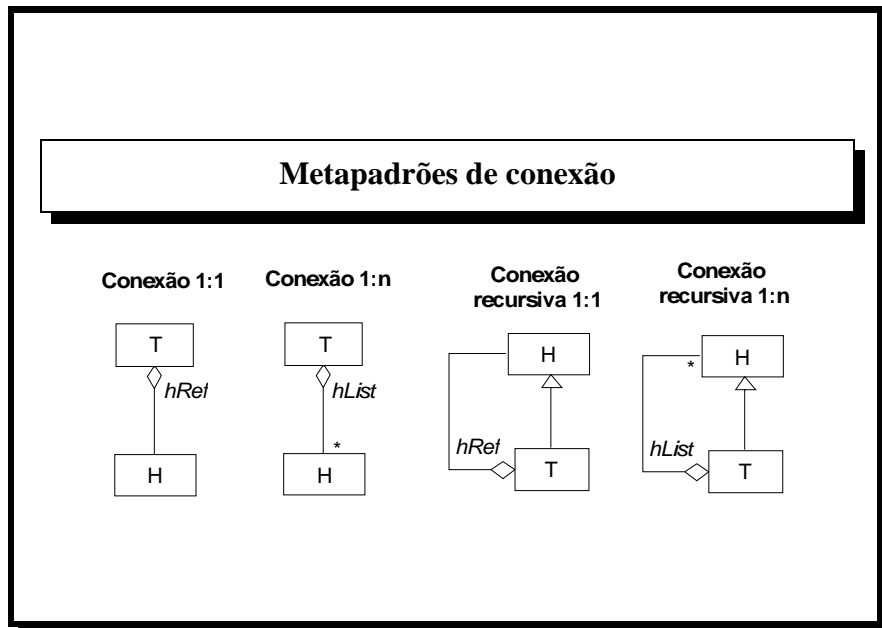
Slide 92

## Metapadrões de unificação

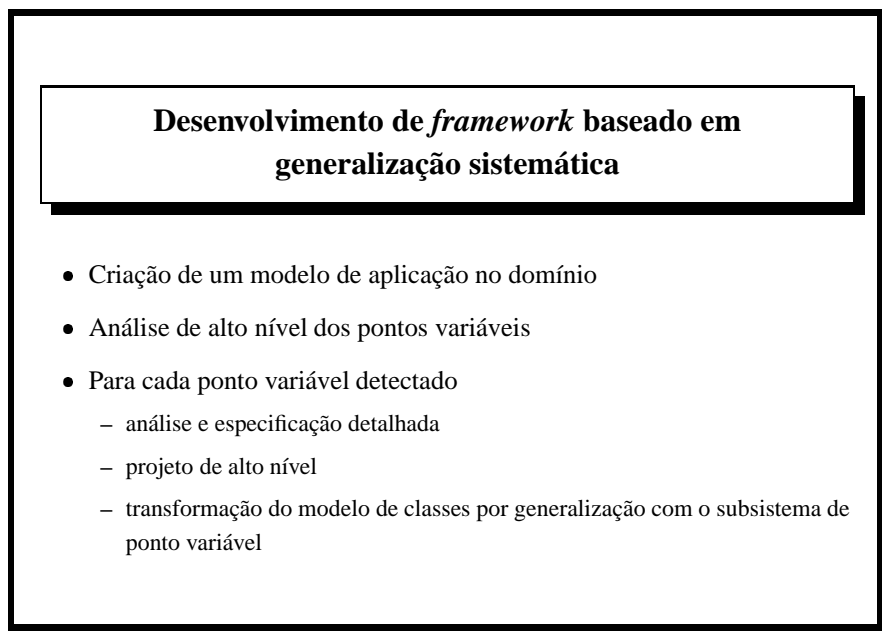
Unificação

Unificação  
recursiva 1:1Unificação  
recursiva 1:n

Slide 93



Slide 94



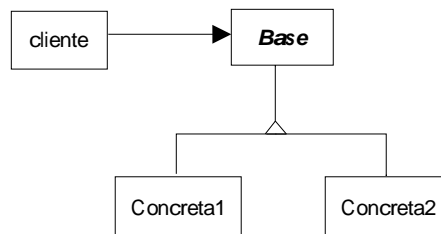
Slide 95

### Subsistemas de ponto variável (*hot spots*)

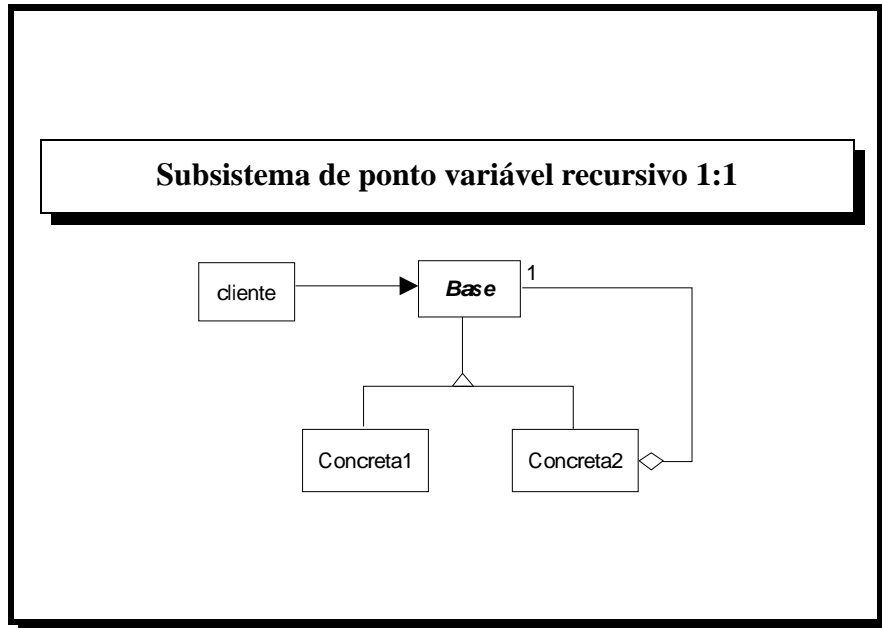
- Outra forma de estabelecer os padrões básicos de relação entre as classes genéricas e suas concretizações
- Variabilidade é obtida por meio de uma referência polimórfica, que estabelece a ligação dinâmica entre o método genérico e as operações concretas
- Quando a ligação faz referência a serviços dentro do próprio sistema, o subsistema de ponto variável é recursivo

Slide 96

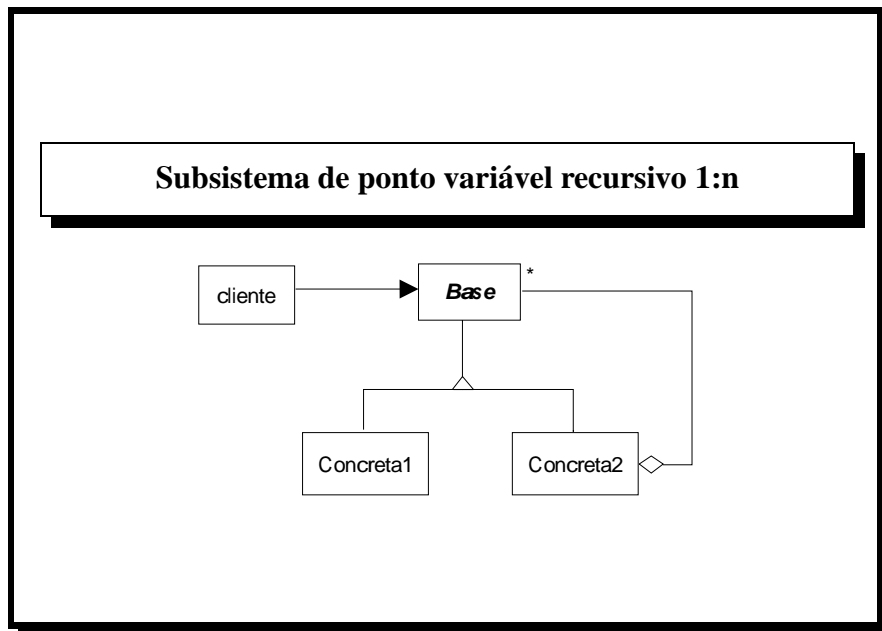
### Subsistema de ponto variável não-recursivo



Slide 97



Slide 98



## Relação entre padrões, metapadrões e subsistemas de ponto variável

Slide 99

Subsistema de ponto variável	Metapadrão	Padrão de projeto
não recursivo	unificação, conexão 1:1, conexão 1:n	fábrica abstrata, método fábrica, protótipo, ponte, comando, iterador, observador, estratégia, método gabarito, construtor, adaptador, mediador, estado, visitante
recursivo 1:1	conexão recursiva 1:1, unificação recursiva 1:1	cadeia de responsabilidade, decorador
recursivo 1:n	conexão recursiva 1:n, unificação recursiva 1:n	composto, interpretador

## Uso do *framework*

Slide 100

- Quando ocorre o desenvolvimento de aplicações
  - Instanciação do *framework*
- Sucesso dependente em grande parte da boa documentação sobre o *framework*
  - Identificação dos pontos de adaptação
  - Padrões utilizados
  - Exemplos
- Aprender a usar um *framework* requer investimento de tempo e dinheiro

## Slide 101

**Manutenção de *frameworks***

- Desenvolvimento de aplicações aumenta reusabilidade do *framework*
  - maior disponibilidade de classes concretas (reuso caixa-preta)
  - identificação de deficiências para futuras extensões
- Revisões em *frameworks* tendem a ser problemáticas
  - compatibilidade de aplicações já desenvolvidas com o mesmo *framework*; devem evoluir juntas
- Em alguns casos, revisão do domínio
  - estabelecimento de novas fronteiras

## Slide 102

**Classificação de *frameworks* pelo escopo de uso**

- Infra-estrutura:** *framework* simplifica o desenvolvimento da infra-estrutura de sistemas de forma portátil e eficiente; tipicamente de uso no desenvolvimento interno às empresas
- Integração *middleware*:** *framework* permite a integração de componentes e aplicações distribuídas; parte importante dos sistemas modernos
- Aplicação empresarial:** *framework* voltado para uma área de aplicação; foco de desenvolvimento nas empresas desenvolvendo aplicações para usuários finais

### Potenciais problemas na integração de *frameworks*

Foco no desenvolvimento dos *frameworks* está na extensão das funcionalidades, não na integração com outros *frameworks*

Slide 103

- Alguns *frameworks* podem assumir que têm controle completo sobre o fluxo de execução da aplicação
  - Quando mais de um *framework* em uso...?
- Como integrar sistemas legados a *frameworks*?
- O que acontece se dois *frameworks* têm componentes com sobreposição de funcionalidades?

### Causas dos problemas de integração de *frameworks*

**Coesão do *framework*:** quão amarrada está a conexão de uma classe do *framework* às demais?

**Cobertura do domínio:** quão bem especificado e isolado está o domínio de aplicação do *framework*?

**Objetivos do projeto:** os desenvolvedores do *framework* devem explicitar se integração foi uma preocupação no projeto

**Falta de acesso ao código fonte:** integração pode requerer modificações e adaptações no código; é importante ter uma abordagem *open source*

**Falta de padrões:** ainda não há padrões voltados para o desenvolvimento de *frameworks*

Slide 104

Slide 105

### Estratégias para integração de *frameworks*

- Usar *threads* de execução independentes para cada *framework*
- Usar padrão Adaptador para estabelecer integração com código legado
- Alterar código do *framework*
- No desenvolvimento:
  - Tornar explícitas e bem documentadas as decisões relativas à arquitetura do *framework*
  - Construir o *framework* em “blocos independentes”
  - Estabelecer no *framework* previsões para configuração, mediação e adaptação, através do uso de padrões

Slide 106

### Aplicando as técnicas de reuso na programação C++

Herança, composição, *templates*

Slide 107

## O conceito da herança

- Princípio da substituição de Liskov
  - Se a classe D é derivada da classe B, quando um objeto B for necessário é possível usar um objeto D
  - D **is-a** B (mas não vice-versa)
- Definição de (boas) hierarquias de classes é um dos conceitos chaves por trás da orientação a objetos

Slide 108

## Aspectos da implementação da herança em C++

- Mecanismos de derivação
  - Implementando hierarquias IS-A
  - Lidando com exceções nas hierarquias
- Herança de interfaces vs herança de implementações
  - separação de interface e implementação
  - redefinições de métodos

## C++ e hierarquias IS-A

Slide 109

- Através da derivação `public`

```
class Pessoa { ... };  
class Estudante : public Pessoa { ... };
```

- Todo estudante é uma pessoa
- Nem toda pessoa é um estudante

Slide 110

```
void dance(const Pessoa& p);  
void estude(const Estudante& e);  
Pessoa p;  
Estudante e;  
dance(p);  
dance(e);  
estude(p);           // oops  
estude(e);
```

## Herança pública vs. herança privada

Slide 111

- Derivação usando `private` **não define** uma hierarquia do tipo IS-A
- como todos os membros da superclasse, públicos ou protegidos, tornam-se privados na nova classe, a interface da superclasse não é herdada
- não há conversão automática de um objeto do tipo derivado para um objeto do tipo da classe base
- falha o princípio da substituição

Slide 112

```
class Pessoa { ... };
class Estudante : private Pessoa { ... };
void dance(const Pessoa& p);
void estude(const Estudante& e);
Pessoa p;
Estudante e;
dance(p);
dance(e);           // oops
estude(p);         // oops
estude(e);
```

Slide 113

## Lidando com pingüins voadores

- “Pássaros podem voar, pingüins são pássaros...”

```
class Passaro {
public:
    virtual void voe();
    ...
};
class Pinguim : public Passaro { ... };
```

- “...mas pingüins não voam!”

Slide 114

## Tratando o problema durante a execução

```
void erro(const string& mens);

class Pinguim : public Passaro {
public:
    virtual void voe() { erro(``Pingüim não voa``);
    ...
};
```

- “Pingüins podem tentar voar, mas fazê-lo é um erro.”
- Abordagem tipicamente adotada em linguagens interpretadas, mas não em C++

## Reverendo a hierarquia de classes

- Se não houver um método `voe` definido para pingüins, compilação já detectará o erro

Slide 115

```
class Passaro { ... };  
class PassaroVoador: public Passaro (  
public: virtual void voe(); ...  
});  
class PassaroNaoVoador: public Passaro { ... };  
class Pinguim: public PassaroNaoVoador { ... };
```

## Potenciais problemas na definição de hierarquias de classes por herança

- Uso de intuição ou “bom senso” pode levar a hierarquias falhas
  - Exemplo com pingüins pode parecer óbvio, mas é emblemático de situações reais de modelagem
- Abuso de herança
  - Nem sempre a relação adequada entre duas classes é a de herança
  - Herança **deve sempre ser** uma expressão da relação IS-A

Slide 116

## Interfaces e implementações

Slide 117

Separação entre a especificação de uma interface e sua implementação é essencial na boa programação orientada a objetos

- Interfaces tendem a estabilizar rapidamente, implementações não
- Programar com base no conhecimento apenas da interface favorece programação genérica e reduz a dependência de compilação entre módulos do sistema

## C++ favorece a mistura de interface e implementação

Slide 118

```
class Pessoa {
public:
    Pessoa(string& nome, Data& aniv, Endereco& end, Pais& p);
    virtual ~Pessoa();
    string nome() const;
    string dataNascimento() const;
    string endereco() const;
    string nacionalidade() const;
private:
    string name_;
    Data birthday_;
    Endereco address_;
    Pais nation_;
};
```

## Slide 119

**Destrutor virtual**

- Por quê?

```
class Base { public: ~Base(); ... };  
class Derivada: public Base { public: ~Derivada(); ... };  
Base *p = new Derivada;  
delete p; // comportamento indefinido
```

- Se destrutor na classe base for declarado como virtual, ambos serão invocados
  - Deve estar presente em qualquer classe que sirva de base para outras
  - Mesmo que virtual puro, deve ter implementação

## Slide 120

**Dependência em relação a outras definições**

- Classe Pessoa usa outras classes na definição de seus atributos

- Tipicamente, no início do módulo:

```
#include <string>  
#include "data.h"  
#include "endereco.h"  
#include "pais.h"
```

- Cria dependência deste módulo (e dos que utilizem a classe Pessoa) em relação àqueles

## Slide 121

**Reduzindo a dependência entre classes**

- Usar declarações ao invés de definições
  - Código como

```
class Data;  
Data hoje();  
void ajustaData(Data d);
```

não precisa conhecer a definição da classe Data para compilar
- Da mesma forma, para definir ponteiros ou referências basta conhecer a declaração, não a definição de um tipo

## Slide 122

**Separando os detalhes de implementação da interface**

```
#include <string>  
class Data; class Endereco; class Pais;  
class PessoaImpl;  
class Pessoa {  
public:  
    Pessoa(string& nome, Data& aniv, Endereco& end, Pais& p);  
    virtual ~Pessoa();  
    string nome() const;  
    string dataNascimento() const;  
    string endereco() const;  
    string nacionalidade() const;  
private:  
    PessoaImpl *parte_impl;  
};
```

## Repassando o trabalho para a implementação

```
#include "Pessoa.h"
#include "PessoaImpl.h"
Pessoa::Pessoa(string& n, Data& d, Endereco& e, Pais& p) {
    parte_impl = new PessoaImpl(n, d, e, p);
}
string Pessoa::nome() const {
    return parte_impl -> getNome();
}
...
```

- Qual padrão de projeto está presente nesta abordagem?

Slide 123

## Definindo interfaces puras (Protocolos)

- Protocolos são classes abstratas sem nenhuma implementação
  - sem construtores
  - sem atributos
  - todos os métodos abstratos (virtuais puros)
- Classes que usam os protocolos operam com ponteiros ou referências para essas classes
  - não é possível instanciar diretamente objetos de protocolos
  - objetos de classes derivadas podem ser criados

Slide 124

Slide 125

### Exemplo de protocolo C++

```
class Pessoa {
public:
    virtual ~Pessoa();
    virtual string nome() const = 0;
    virtual string dataNascimento() const = 0;
    virtual string endereco() const = 0;
    virtual string nacionalidade() const = 0;
};
```

Slide 126

### Construção de objetos associados a protocolos

- Objetos serão de classes derivadas do protocolo

```
class PessoaMesmo: public Pessoa {
public:
    PessoaMesmo(string& n, Data& d, Endereco& e, Pais& p)
        : name_(n), birthday_(d), address_(e), nation_(p) {}
    string nome() const;
    ...
private:
    string name_;
    ...
};
```

## Construção de objetos via protocolos

- Em geral, dá-se através de método estático associado ao próprio protocolo

```
class Pessoa {
public:
    ...
    static Pessoa * fazPessoa(string& n, Data& d,
                               Endereco& e, Pais& p);
};
Pessoa * Pessoa::fazPessoa(string& n, Data& d,
                             Endereco& e, Pais& p) {
    return new PessoaMesmo(n, d, e, p);
}
```

- Algum padrão reconhecido aqui?

Slide 127

## Herança de interface e herança de implementação de métodos

- O que de um método se pretende passar de uma classe base para uma classe derivada?
  - Apenas sua interface (declaração);
  - A interface e a implementação, porém permitindo que classe derivada defina nova implementação (especializações);
  - A interface e a implementação, sem permitir que a classe derivada defina nova implementação (aspecto invariante).

Slide 128

Slide 129

## Três tipos de métodos

- Para a classe abstrata Shape,

```
class Shape {  
public:  
    virtual void draw() = 0;  
    virtual void error(string & msg);  
    int objectID();  
    ...  
};
```

os três métodos estarão presentes em suas classes derivadas publicamente.

Slide 130

## Apenas herança de interface

- Obtida pelo uso da função virtual pura
  - No exemplo, draw
  - Não precisa (mas até poderia) ter uma implementação
- Se Rectangle e Ellipse são derivadas de Shape,

```
Shape *ps1 = new Rectangle; ps1->draw();  
Shape *ps2 = new Ellipse; ps2->draw();  
ps1->Shape::draw();
```

## Slide 131

**Herança de interface com implementação padrão**

- Obtida com métodos virtuais “normais”
- Classe derivada pode optar entre usar a implementação padrão oferecida pela superclasse ou definir a própria implementação, especializada
- Potencial problema de manutenção
  - Se a hierarquia crescer (novas classes derivadas) e o padrão não mais for aplicável

## Slide 132

“Não há diferença entre pilotar um Avião ModeloA ou um Avião ModeloB; outros poderiam ser diferentes:”

```
class Aviao {  
public:  
    virtual void voePara(string& destino);  
    ...  
};  
class ModeloA : public Aviao { ... };  
class ModeloB : public Aviao ( ... );
```

## Slide 133

“Já pilotar o novo Avião ModeloC é diferente...”

```
class ModeloC : public Aviao { ... //oops};  
Aviao eqp = new ModeloC;  
eqp->voePara("JFK"); // desastre
```

- Problema não é ter uma implementação padrão, mas permitir que a classe derivada a utilize sem dizer explicitamente que irá fazê-lo

### Separando a interface da implementação padrão

- Usar método privativo, não-virtual:

```
class Aviao {  
public:  
    virtual void voePara(string& dest) = 0;  
    ...  
private:  
    void vaiPorMim(string& dest);  
};  
void ModeloA::voePara(string& dest) {  
    vaiPorMim(dest);  
}
```

## Slide 134

## Separando a interface da implementação padrão (2)

- Alternativamente, pode usar definição para função virtual pura:

```
class Aviao {
public:
    virtual void voePara(string& dest) = 0;
    ...
};
void Aviao::voePara(string& dest) {
    // procedimento padrão
}
void ModeloA::voePara(string& dest) {
    Aviao::voePara(dest);
}
```

- Perde a “proteção” para implementação padrão

Slide 135

## Métodos invariantes na especialização

- Comportamento **não pode** ser alterado em classes derivadas
- Situação obtida com uso das funções não-virtuais
  - Define interface e implementação mandatória
- Método não-virtual nunca deve ser redefinido em classes derivadas

Slide 136

## Slide 137

**Nunca redefinir métodos não-virtuais**

```
class B { public: void mf(); ... };  
class D: public B { ... };  
D x;  
B *pB = &x; pB->mf();  
D *pD = &x; pD->mf();
```

- Se D redefina mf, as duas invocações de mf terão comportamento diferente

## Slide 138

**Porque não redefinir métodos não-virtuais**

- Métodos não virtuais são ligados em tempo de compilação
- Mesmo que através de ponteiros, a implementação utilizada será a do tipo do ponteiro e não a do tipo do objeto na execução
- O mesmo ocorre para valores padrão de funções virtuais
  - valores padrão são ligados estaticamente
  - pode invocar corpo definido na classe derivada com valores padrões definidos na superclasse
  - conclusão: não devem ser redefinidos

Slide 139

## Trabalhando com composição

- Para modelar expressões do tipo **tem um** (objeto de outra classe) ou **é implementado usando** (outro tipo de objeto)

```
class Pessoa {  
    ...  
private:  
    string nome; // Pessoa tem um nome  
    Endereco end; // e tem um endereco  
    ...  
};
```

Slide 140

## Composição e “é implementado usando”

- Exemplo: implementar uma coleção do tipo conjunto usando uma estrutura de dados do tipo lista
- um conjunto não é uma lista
  - conjuntos não podem ter elementos duplicados, listas podem
  - herança pública não é uma boa opção
- pode definir atributo usando `list` da STL

**Slide 141**
**Definição da classe Set:**

```

template<class T>
class Set {
public:
    bool member(const T& item) const;
    void insert(const T& item);
    void remove(const T& item);
    int cardinality( ) const;
private:
    list<T> rep;
};
    
```

**Slide 142**
**Exemplo de métodos de Set:**

```

template<class T>
bool Set<T>::member(const T& item) const {
    return find(rep.begin(), rep.end(), item) != rep.end();
}
template<class T>
void Set<T>::insert(const T& item) {
    if(!member(item)) rep.push_back(item);
}
template<class T>
void Set<T>::remove(const T& item) {
    list<T>::iterator it = find(rep.begin(), rep.end(), item);
    if (it != rep.end()) rep.erase(it);
}
template<class T>
int Set<T>::cardinality( ) const {
    return rep.size();
}
    
```

## Herança ou *templates*

Slide 143

- Na definição de um conjunto de classes com pequenas diferenças entre si, qual mecanismo usar?  
**herança:** fatorar parte comum em superclasse (abstrata), derivar as classes concretas e em cada uma definir a especialização da classe  
**template:** definir o código para a classe genérica em termos de um tipo parametrizado e instanciar, para cada tipo desejado, uma versão da definição da classe especializada
- Usar herança quando o tipo do objeto muda o comportamento dos métodos, usar *template* quando comportamento é invariante com o tipo parametrizado

Slide 144

```
template<class T> class Stack {
public:
    Stack();
    ~Stack();
    void push(const T& obj);
    T pop();
    bool empty() const;
private:
    struct StackNode {
        T data;
        StackNode *next;
        StackNode(const T& newData, StackNode *nextNode)
            : data(newData), next(nextNode) { }
    };
    StackNode *top;
    Stack(const Stack& rhs);
    Stack& operator=(const Stack& rhs);
};
```

## Slide 145

```
[--- template<class T> ---]
Stack<T>::Stack() : top(0) {}
void Stack::push(const T& obj) {
    top = new StackNode(obj, top); }
T Stack<T>::pop() {
    StackNode *topOfStack = top;
    top = top->next;
    T data = topOfStack->data;
    delete topOfStack;
    return data; }
Stack<T>::~~Stack() {
    while(top) {
        StackNode *toDie = top;
        top = top->next;
        delete toDie;
    } }
bool Stack<T>::empty() const {
    return top == 0; }
```

## Slide 146

***Templates vs ponteiros genéricos***

- Ponteiros genéricos (`void*`) oferecem outra alternativa que permite ter o mesmo comportamento para diferentes tipos de objetos
  - sem a replicação de código que *templates* geram
- Dois passos
  1. criar a classe que manipula os ponteiros genéricos
  2. criar classes que enforçam a conversão correta dos ponteiros

## Slide 147

Classe pilha genérica (opção 1):

```
class GenericStack {
public:
    GenericStack();
    ~GenericStack();
    void push(void *obj);
    void *pop();
    bool empty() const;
private:
    struct StackNode {
        void *data;
        StackNode *next;
        StackNode(void *newData, StackNode *nextNode)
            : data(newData), next(nextNode) { } };
    StackNode *top;
    GenericStack(const GenericStack& rhs);
    GenericStack& operator=(const GenericStack& rhs); };
```

## Slide 148

Classe para conversão de tipos (opção 1):

```
class IntStack {
public:
    void push(int *ip) {s.push(ip);}
    int *pop() {return static_cast<int *>(s.pop());}
    bool empty() const { return s.empty();}
private:
    GenericStack s;
};
```

### Comentários sobre essa implementação

#### Slide 149

- Não há custo adicional no uso de `IntStack`
  - todos os métodos são *inline*
- Nada impede que cliente use diretamente objetos da classe `GenericStack`
  - Alternativa: evitar manipulação direta de `GenericStack` protegendo sua criação e interface

#### Classe pilha genérica (opção 2):

```
class GenericStack {
protected:
    GenericStack();
    ~GenericStack();
    void push(void *obj);
    void *pop();
    bool empty() const;
private:
    struct StackNode {
        void *data;
        StackNode *next;
        StackNode(void *newData, StackNode *nextNode)
            : data(newData), next(nextNode) { } };
    StackNode *top;
    GenericStack(const GenericStack& rhs);
    GenericStack& operator=(const GenericStack& rhs); };
```

#### Slide 150

**Slide 151**

Classe para conversão de tipos (opção 2):

```
class IntStack: private GenericStack {
public:
    void push(int *ip) {GenericStack::push(ip);}
    int *pop() {return static_cast<int *>(GenericStack::pop());}
    bool empty() const { return GenericStack::empty();}
};
```

**Slide 152**

Classe para conversão de tipos (generalizando os tipos possíveis de interface):

```
template<class T>
class Stack: private GenericStack {
public:
    void push(T *op) {GenericStack::push(op);}
    T *pop() {return static_cast<T*>(GenericStack::pop());}
    bool empty() const { return GenericStack::empty();}
};
```

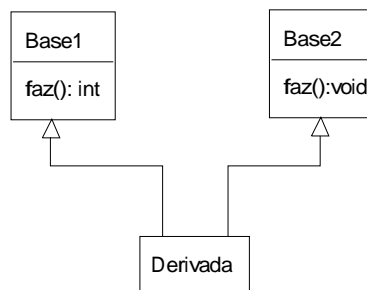
Slide 153

## Herança múltipla

- Não há uma posição clara na comunidade de orientação a objetos sobre os benefícios da herança múltipla
- Por que deve ser evitada (ou, pelo menos, usada com muito cuidado...)?
  - Ambigüidade potencial
  - Múltiplas ocorrências da base

Slide 154

## Ambigüidade potencial



## Ambigüidade potencial (C++)

Slide 155

```
class Base1 {          class Base2 {
public:                public:
    int faz();         void faz();
};                    };

class Derivada: public Base1, public Base2 {
... // sem faz()
};

Derivada d;
d.faz();              // erro de compilação
```

Restringir acesso não resolve o problema:

Slide 156

```
class Base1 {          class Base2 {
public:                private:
    int faz();         void faz();
};                    };

class Derivada: public Base1, public Base2 {
... // sem faz()
};

Derivada d;
int i = d.faz();      // continua erro de compilação
```

**Slide 157**

Por que compilador não leva em conta as especificações de acesso?

- Porque modificações na visibilidade de membros das classes não deveria modificar o significado de programas
- Se a abordagem anterior “resolvesse” o problema, apenas modificação na visibilidade mudaria o comportamento do programa
  - sem modificar invocação ou corpo dos métodos

**Slide 158**

Para resolver ambigüidade, apenas através da qualificação dos membros (referências explícitas à classe base):

```
d.Base1::faz();  
d.Base2::faz();
```

- Se métodos fossem virtuais, não teria como explorar redefinições
  - mesmo que objeto fosse de uma classe derivada da base, a qualificação faz com que o método invocado seja exatamente o especificado.

## Slide 159

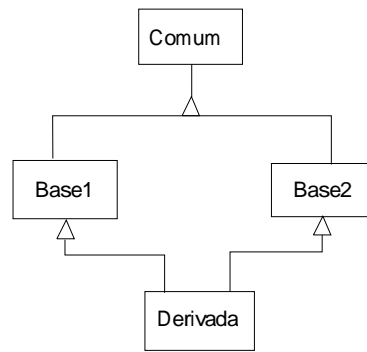
E se dois métodos, com os mesmos tipos de argumentos, fossem virtuais e a classe derivada quisesse redefinir os dois?

- não há como fazer diretamente
  - Apenas um método pode existir numa classe com um dado nome e lista de tipos de argumentos
- possível através da criação de classes auxiliares, sem correspondência com a modelagem da aplicação

## Slide 160

```
class AuxBase1: public Base1 {
public:
    virtual int faz1() = 0;
    virtual int faz() { return faz1(); }
};
class AuxBase2: public Base2 {
public:
    virtual void faz2() = 0;
    virtual void faz() { faz2(); }
};
class Derivada: public Base1, public Base2 {
public:
    virtual int faz1();
    virtual void faz2();
};
Derivada *d = new Derivada;
Base1 *p1 = d; Base2 *p2 = d;
p1->faz(); // chama faz1
p2->faz(); // chama faz2
```

## Múltiplas ocorrências da classe base



Slide 161

Tipicamente, classe comum é uma classe base virtual:

```
class Comum { ... };
class Base1: virtual public Comum { ... };
class Base2: virtual public Comum { ... };
class Derivada: public Base1, public Base2 { ... };
```

Slide 162

- Classe base virtual impõe penalidades de acesso (tipicamente, implementação por ponteiros na estrutura interna)
- Mas e se Comum, Base1 e Base2 existissem antes e independentemente de Derivada (por exemplo, em uma biblioteca)?
  - bom projeto requer visão profética. . .

## Outros problemas na herança múltipla

Passagem de argumentos para construtor de classe base virtual:

- Em herança simples, sem bases virtuais, construtores de classe em nível  $i$  repassam informação para construtores da classe no nível  $i - 1$
- Na herança múltipla, lista de inicialização do construtor está na classe que é mais derivada da base
  - pode estar distante na hierarquia de classes
  - pode variar a posição com evolução da hierarquia

**Solução:** classes bases virtuais sem atributos (estilo *interface* de Java)

Slide 163

Dominância das funções virtuais:

```
class Comum { public: virtual void f(); ... };  
class Base1: virtual public Comum { ... };  
class Base2: virtual public Comum {  
    public: virtual void f(); ... };  
class Derivada: public Base1, public Base2 { ... };
```

- Há ambigüidade nesse código?  

```
Derivada *pd = new Derivada;  
pd->f();
```
- Se Comum não fosse base virtual de Base1 ou Base2, haveria
- Como é,  $f$  da Base2 é utilizada (domina a hierarquia)

Slide 164

Slide 165

### Como usar herança múltipla de forma segura?

- Evitar grafos de hierarquia na forma de diamantes
  - Evita problemas associados a classes bases virtuais
- É seguro combinar herança pública de interface com herança privada de implementação
- Rever hierarquia: herança múltipla é mesmo a melhor solução?

Slide 166

### Usando C++ para expressar padrões de projeto

## Atividades

Analisar as implementações fornecidas em C++ com usos dos padrões de projeto *Adaptador*, *Decorador*, *Mediador*, *Singleton* e *TemplateMethod*. Para cada um deles, descreva

Slide 167

1. Qual o problema que está sendo abordado;
2. Que alternativas de implementação são consideradas;
3. Que construções de C++ são relevantes para a implementação;
4. Quais os potenciais problemas ou deficiências da implementação;
5. Se as técnicas indicadas poderiam ser úteis na implementação de outros padrões.

## Conclusões

Slide 168

**Slide 169**

- Origem dos maiores problemas na programação em C++
  - Falta de compreensão do paradigma de orientação a objetos
  - Vícios no desenvolvimento de *software*
  - Problemas no projeto
  - Mal uso dos recursos da linguagem
- Para o último item, solução passa por
  - Reconhecer as “mensagens implícitas” associadas a cada recurso
  - Estar atento às situações de risco
  - Aplicar soluções reconhecidas

**Referências e sugestões de leitura****Slide 170**

1. Frederick P. Brooks, Jr. No Silver Bullet: Essence and accidents of software engineering. *IEEE Computer*, pp.10–19, April 1987. Disponível em <http://www.virtualschool.edu/mon/SoftwareEngineering/BrooksNoSilverBullet.html>.
2. Phillip G. Armour. The Five Orders of Ignorance. *Communications of the ACM* 43(10), pp.17–20, October 2000.
3. William H. Brown, Raphael C. Malveau, Hays W. McCormick III, and Thomas J Mowbray. *Antipatterns: Refactoring software, architectures, and projects in crisis*. John Wiley & Sons, 1998.
4. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of reusable object-oriented software*. Addison-Wesley, 1995.

**Slide 171**

5. Ralph E. Johnson. Frameworks=Components+Patterns. *Communications of the ACM* 40(10), pp.39–42, October 1997.
6. W. Pree. *Design patterns for object-oriented software development*. Addison-Wesley, 1995.
7. H. A. Schmid. Framework Design by Systematic Generalization. *Building Application Frameworks: Object-Oriented Foundations of Framework Design*, Mohamed E. Fayad, Douglas C. Schmidt e Ralph E. Johnson (Eds.). John Wiley & Sons, 1999, Cap. 15, pp.353–378.
8. Scott Meyers. *Effective C++: 50 specific ways to improve your programs and designs*, 2nd edition. Addison-Wesley, 1998.
  - <http://www.antipatterns.com/>
  - <http://hillside.net/patterns/>
  - [http://www.objenv.com/cetus/oo\\_patterns.html](http://www.objenv.com/cetus/oo_patterns.html)
  - <http://www.osdn.org/>
  - <http://sourceforge.net/>