Teste de Software OO

Fundamentos de Testes de Software

PUC Minas – São Gabriel

Roteiro da Apresentação

- Características do Software OO
- Vantagens do Software OO
- Problemas no Teste de Software OO
- Níveis de Teste de software OO
- Teste de Estado de Binder

Software OO

- sociedade de agentes que cooperam entre si
- estado dos agentes é afetado pela seqüência de mensagens e respostas que uma classe aceita e/ou gera
- alta interação entre componentes (objetos)
- comportamento individual e coletivo

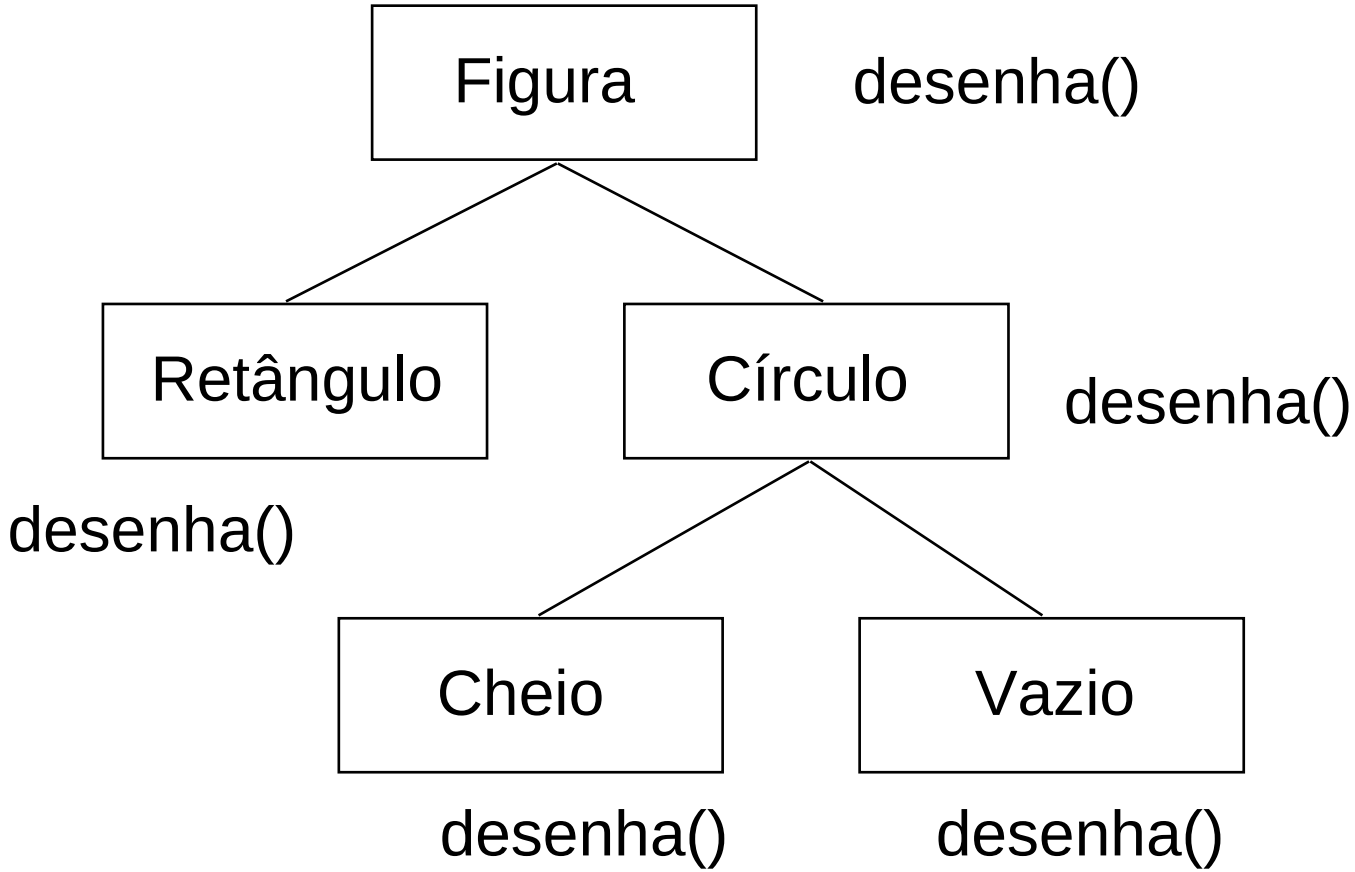
Vantagens do Software OO

- separação explícita entre especificação e implementação de classes
  - interfaces bem-definidas e explícitas
  - testar a especificação -> teste funcional
  - testar a implementação -> teste estrutural
- herança fornece diretivas inerentes para a reutilização de casos de teste

Problemas no Teste de Software OO

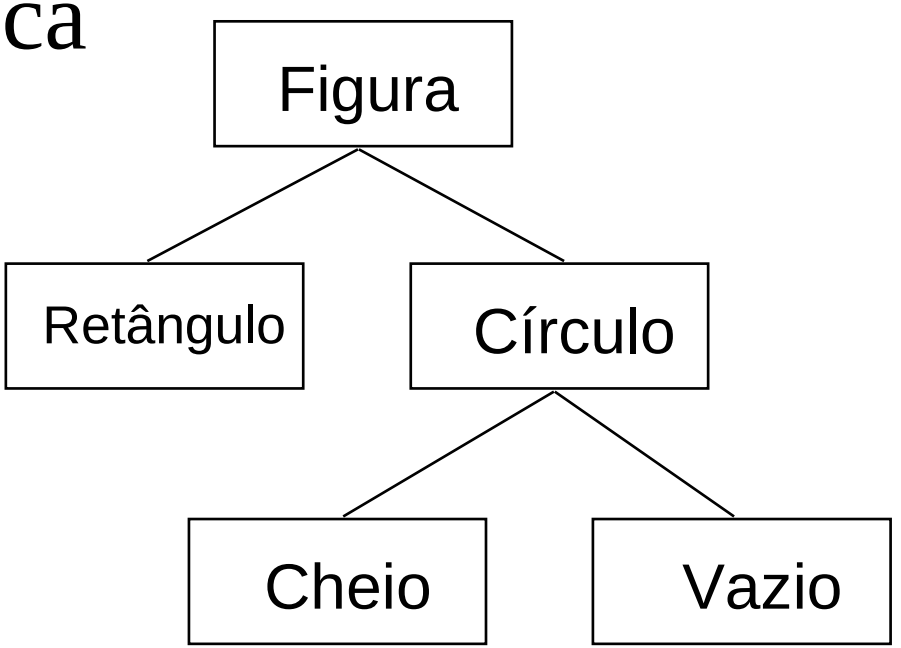
- alta modularidade
  - testar número elevado de componentes
- múltiplos pontos de entrada em uma classe
- elevada interação entre componentes
  - procedimental -> chamadas a subrotinas
  - objetos -> envio de mensagens (+ freqüente)
- polimorfismo e ligação dinâmica expandem as possíveis interações entre objetos

Herança e Polimorfismo



Ligação Dinâmica

```
Figura fig;
. . .
fig = new Retangulo;
. . .
fig = new Circulo();
. . .
fig.desenha();
```



desenha()

fig

desenha() de Retangulo?

desenha() de Círculo?

Erros usualmente encontrados:

- erros em classes
  - estados de objetos definidos na especificação mas não implementados
  - comportamentos definidos na especificação mas não implementados
  - implementação da classe não reflete requisitos

Erros usualmente encontrados:

- erros em mensagens
  - parâmetros impróprios
  - receptores não retornam valores apropriados aos objetos que enviaram a mensagem
- erros em métodos
  - pós-condições não satisfeitas
  - exceções não implementadas
  - tempo de resposta insuficiente (tempo-real)
  - código não executável

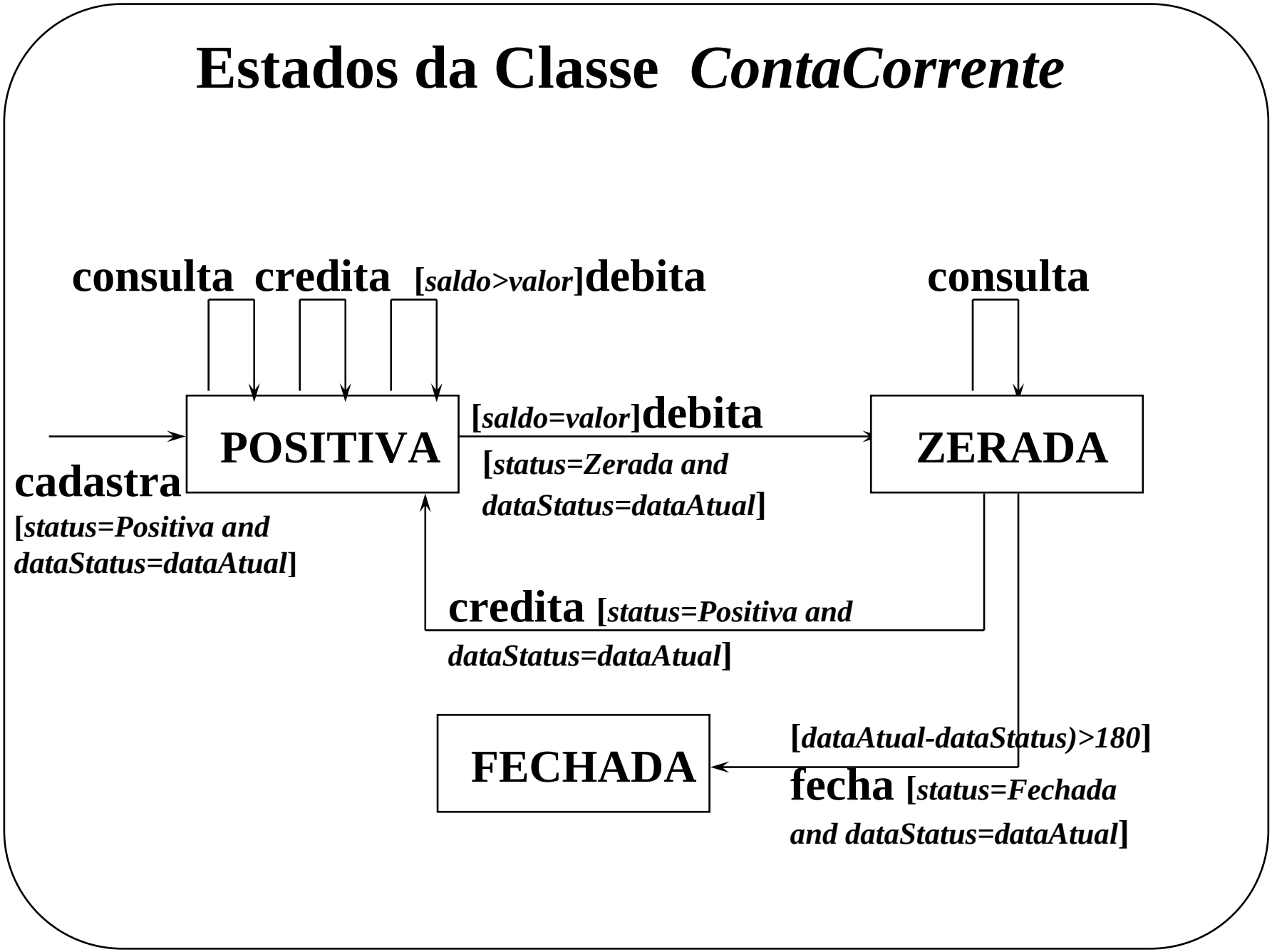
Níveis de Teste de Código OO

- teste de classe
  - atributos + comportamento altamente coesos
  - classe é a menor unidade a ser testada
  - teste de unidade + teste de integração
- teste de cluster (categoria)
  - cluster - conjunto de classes relacionadas que interagem entre si
  - testar interações entre classes -> teste de integração
- teste de sistema
  - testar a funcionalidade da aplicação

Teste de Classe

- tipos de teste
  - teste estrutural dos métodos (teste de caminho)
  - teste baseado em estados do objeto
  - teste de integração de métodos da classe
  - teste funcional da classe (drivers, stubs)

Estados da Classe *ContaCorrente*

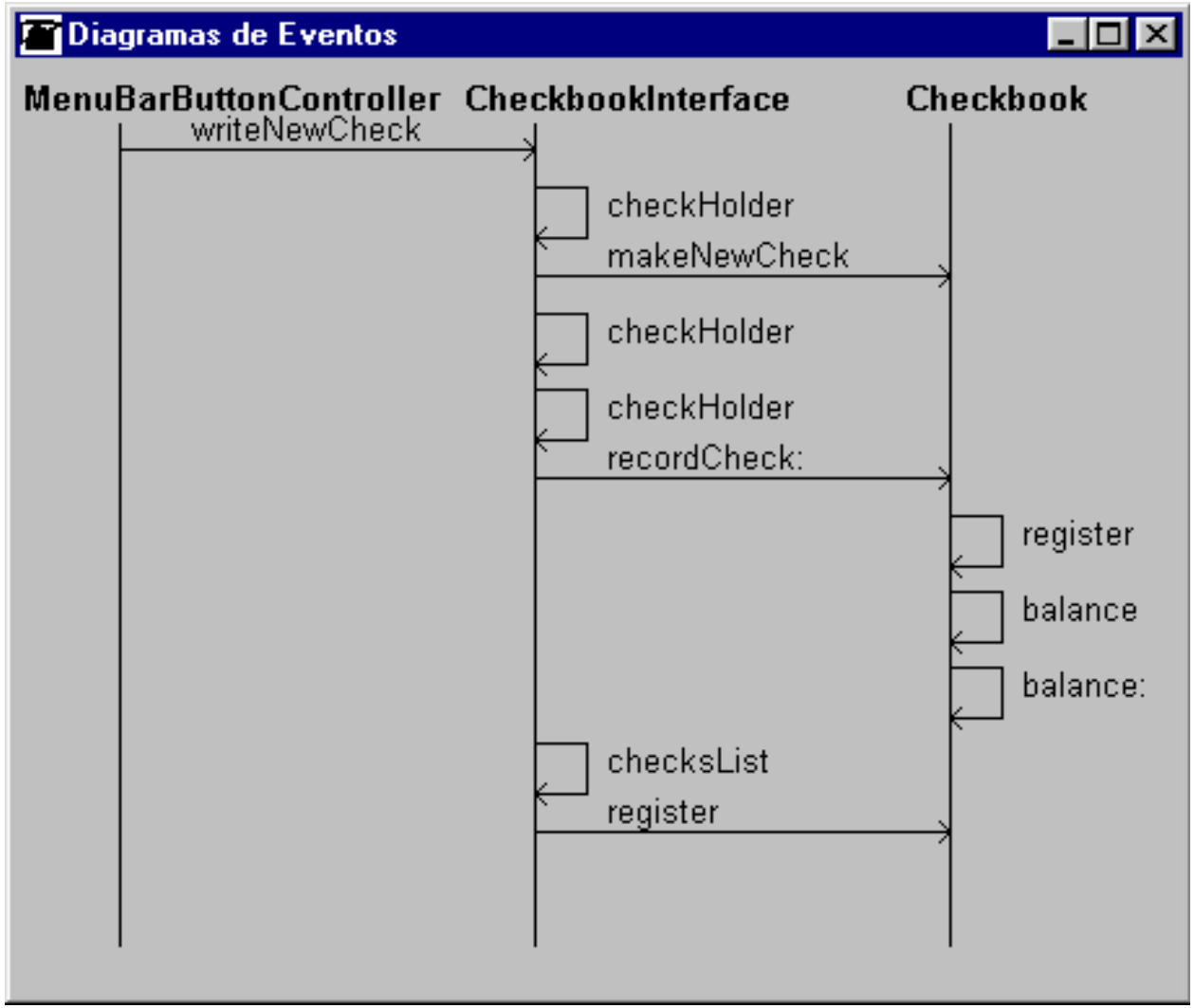


Teste Funcional de Classes

- seleção de casos a partir da especificação da classe
- cobertura expressa pela porcentagem de transições de estados de objetos ou pré-condições de métodos
- pré-condições estabelecem ambiente para teste do objeto
  - gerar casos para cláusulas AND e OU
  - gerar casos para limites das condições
- classes abstratas
  - testar apenas métodos concretos
  - métodos abstratos são testados nas subclasses

Teste de Integração

- Teste de Integração na classe
  - métodos que acessam os mesmos atributos
  - métodos que chamam métodos da própria classe
  - diagrama de estados do objeto identifica as interações dentro da classe
  - cobertura medida através da porcentagem de interações testadas
- Teste de Integração entre classes
  - casos de teste identificados através de mensagens entre classes
  - diagrama de interação entre classes (**cenários, diagrama de eventos**)


Diagramas de Eventos
MenuBarButtonController
CheckbookInterface
Checkbook
writeNewCheck
checkHolder
makeNewCheck
checkHolder
checkHolder
recordCheck:
register
balance
balance:
checksList
register

Teste de Subclasses

- geração de casos deve iniciar a partir da raiz da hierarquia de classes e prosseguir para as subclasses
- permite a reutilização de casos visto que atributos e métodos são herdados das superclasses
- casos de teste são expandidos a medida que a interface da classe aumenta
- nenhum método herdado da superclasse pode ser eliminado na subclasse
- novos estados e transições podem ser adicionados

Teste de Subclasses

- cada método da subclasse é classificado como:
  - herdado: implementado na superclasse
  - redefinido: reimplementado na subclasse
  - novo: implementado apenas na subclasse
- havendo redefinição de métodos
  - casos de teste funcional são sempre aproveitados
  - novos casos de teste estrutural devem ser gerados
- geração de casos de teste para métodos:
  - herdados: desnecessária (**não há interação c/redefinidos**)
  - redefinidos: casos adaptados ou reutilizados
  - novos: geração de casos novos

Teste de Cluster (categoria)

- cluster = conjunto de classes relacionadas que interagem entre si
- interface do cluster = mensagens que são enviadas ao cluster vindas do ambiente externo
- identificar pré e pós-condições para cada método da interface
- gerar casos
  - para o teste funcional: a partir da interface
  - para o teste estrutural: a partir do fluxo de controle e/ou de dados

Seleção de casos de teste [McGregor]

- desenvolver casos de teste:
  - para teste funcional, que cubram a especificação completa da classe
    - ex: classe Empregado, funcionalidades: contratar, demitir, pagar,etc
  - para teste baseado em estados, tal que todas as transições do modelo dinâmico sejam cobertas
    - ex: ContaBancária, estados: ativa, negativa, bloqueada, etc
  - para teste estrutural para cobertura de todas as linhas de código
  - para teste de integração, que cubram as interações entre métodos da mesma classe
  - de teste de integração, que cubram as interações entre objetos da classe sendo testada c/outras classes
    - ex: integração de classes de interface com classes do domínio da aplic.

Verificação de Teste Completo

[McGregor]

- testar objetos criados através de cada método construtor da classe
- testar pares de métodos auxiliares (ex: “get” e “set”)
- testar cada pós-condição de método modificador (de atributos ou estado)
- testar caminhos através do código
- testar combinações de métodos que acessam os mesmos atributos
- testar se cada objeto libera a memória que ocupou

Teste de Estados de Binder

- Base: modelo dinâmico de objetos
- Estado:
  - subconjunto do conjunto de todas as combinações possíveis dos valores dos atributos da classe
- Exemplo: conta bancária

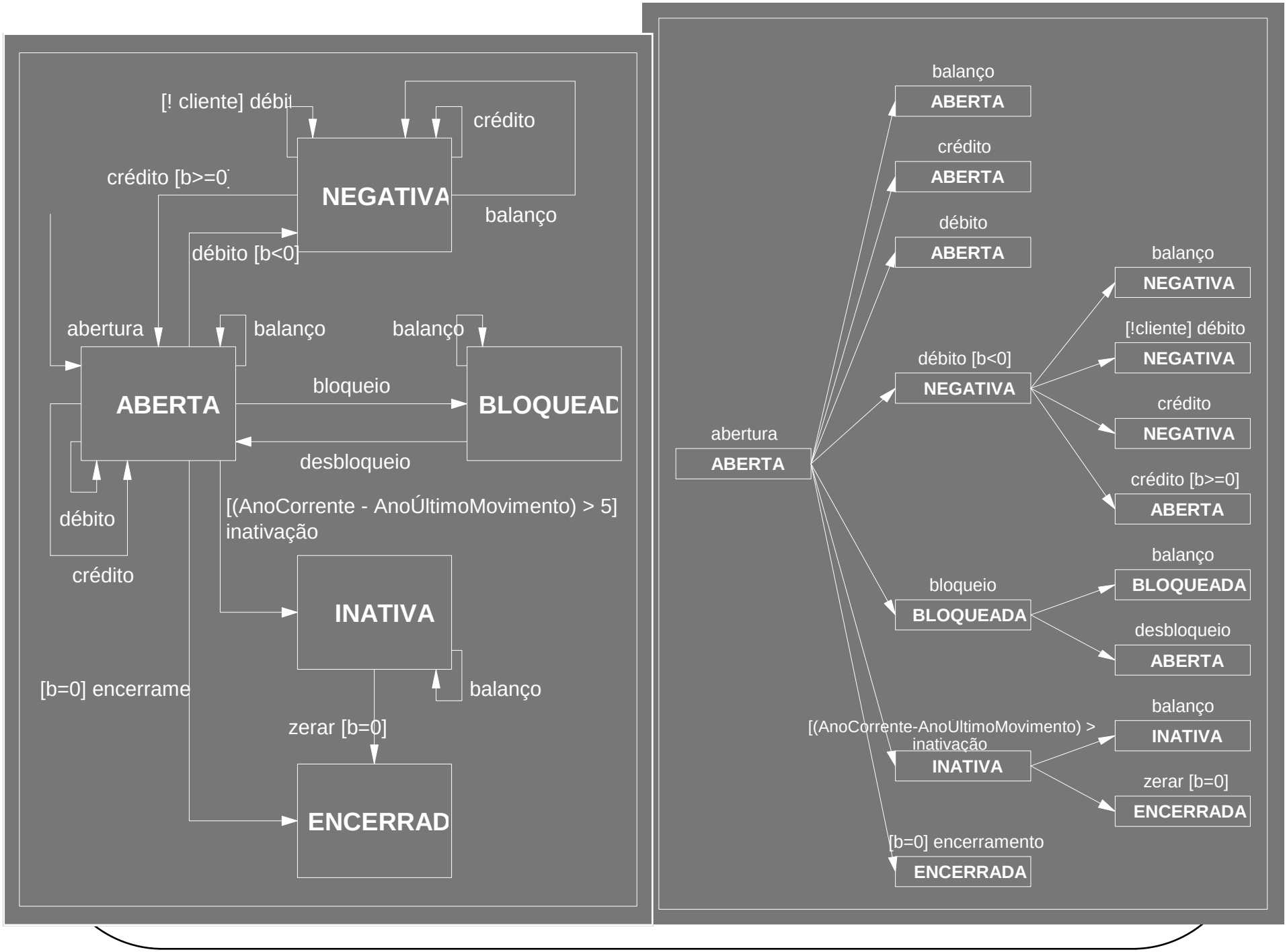
  estados: ativa, negativa, bloqueada, encerrada, etc.
- Contrato da classe: pré e pós-condições de métodos
- Pré-condição: **para que a mensagem seja aceita**.
- Pós-condição: **define estado do objeto após execução.**

Teste de Estado [Binder]

- intervalos de valores de atributos determinam estados diferenciados de objetos
- comportamento da classe leva os objetos a assumirem estados distintos
- evita testar todas as combinações de possíveis valores de atributos

Teste de Estados [Binder]

- comportamento de cada classe é representado por uma máquina de estados finitos.
- cada estado é determinado por atributos do objeto
- uma transição é determinada pela mensagem enviada
- passos no método de Binder
  - especificar a máquina de estados
  - derivar a árvore de transições
  - determinar casos de teste que cubram a árvore de transições


[! cliente] débit
crédito
crédito [b>=0]
NEGATIVA
balanço
débito [b<0]
abertura
balanço
balanço
ABERTA
bloqueio
BLOQUEAD
desbloqueio
débito
crédito
[(AnoCorrente - AnoÚltimoMovimento) > 5]
inativação
INATIVA
[b=0] encerrame
balanço
zerar [b=0]
ENCERRAD
balanço
ABERTA
crédito
ABERTA
débito
ABERTA
abertura
ABERTA
débito [b<0]
NEGATIVA
balanço
NEGATIVA
[!cliente] débito
NEGATIVA
crédito
NEGATIVA
crédito [b>=0]
ABERTA
bloqueio
BLOQUEADA
balanço
BLOQUEADA
desbloqueio
ABERTA
[(AnoCorrente-AnoÚltimoMovimento) >
inativação
INATIVA
balanço
INATIVA
zerar [b=0]
ENCERRADA
[b=0] encerramento
ENCERRADA

Máquina de Estados

- nodos = estados que o objeto pode assumir
  - exemplo: conta aberta, negativa, bloqueada
  - cada estado correspondente a um conjunto de valores de atributos
- arcos = transições de estado = mensagens
  - expressões entre colchetes indicam condições para que a transição ocorra:
    - pré-condição: antes do nome do método
    - pós-condição: após o nome do método

MEF para teste de Binder

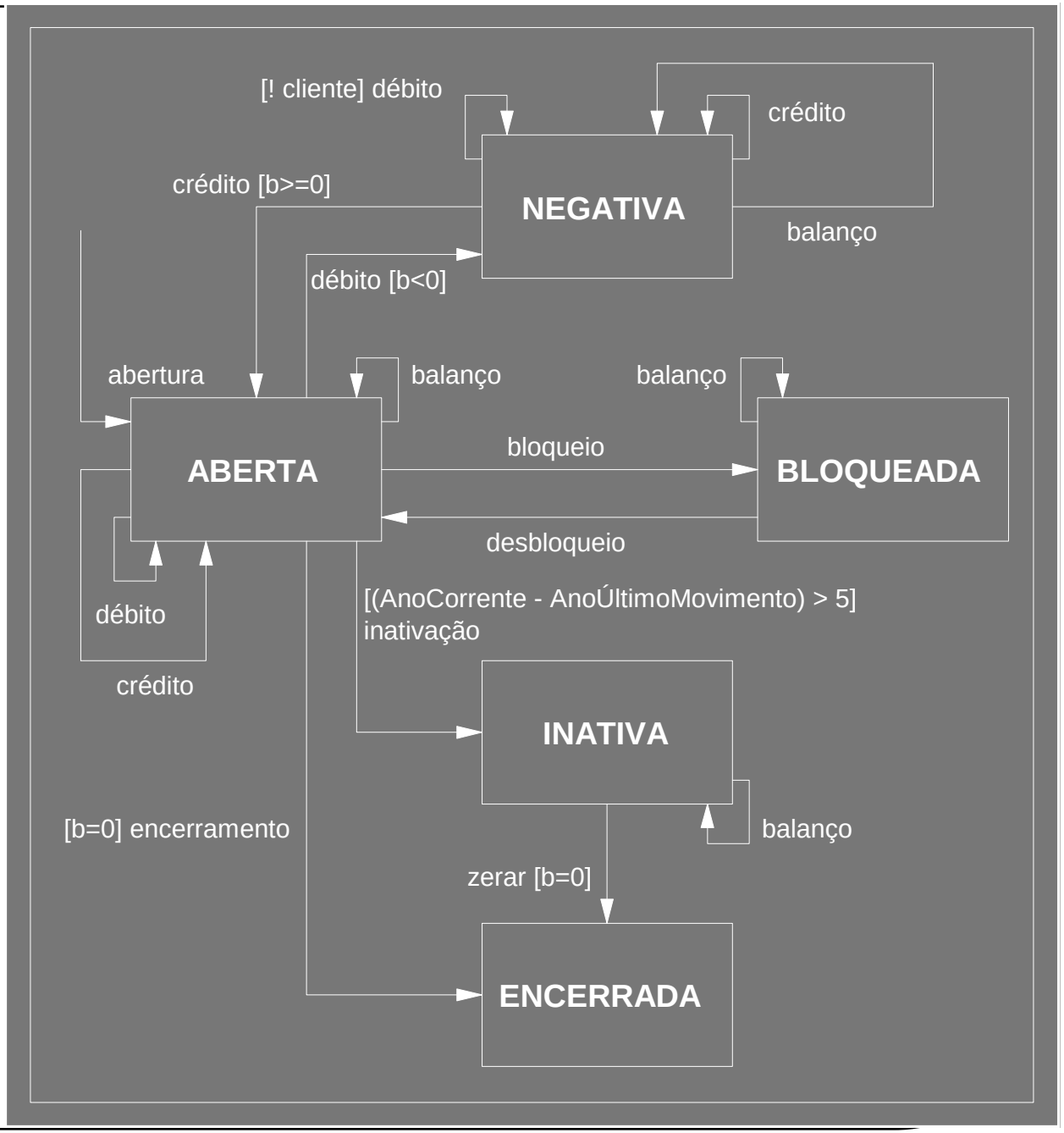
- máquina de estados finitos
  - completa com todos eventos e estados
- máquina de estados mínima
  - sem estados redundantes e desnecessários
- indicar estado inicial
- todos os estados devem ser alcançáveis

Exemplo: Conta Bancária

Nome: **Conta**
Superclasse: **Object**
Atributos:
Saldo: saldo atual da conta
Número: número da conta
ÚltimaTransação: data da realização da última transação
Operações:
Balanço: fornece o saldo atual da conta
Crédito: adiciona valor de crédito à conta
Débito: subtrai valor de débito da conta
Abertura: cria uma conta
Bloqueio: suspende transações na conta
Desbloqueio: suspende bloqueio
Zerar: zera o saldo da conta
Encerramento: finaliza as atividades da conta

Máquina de Estados

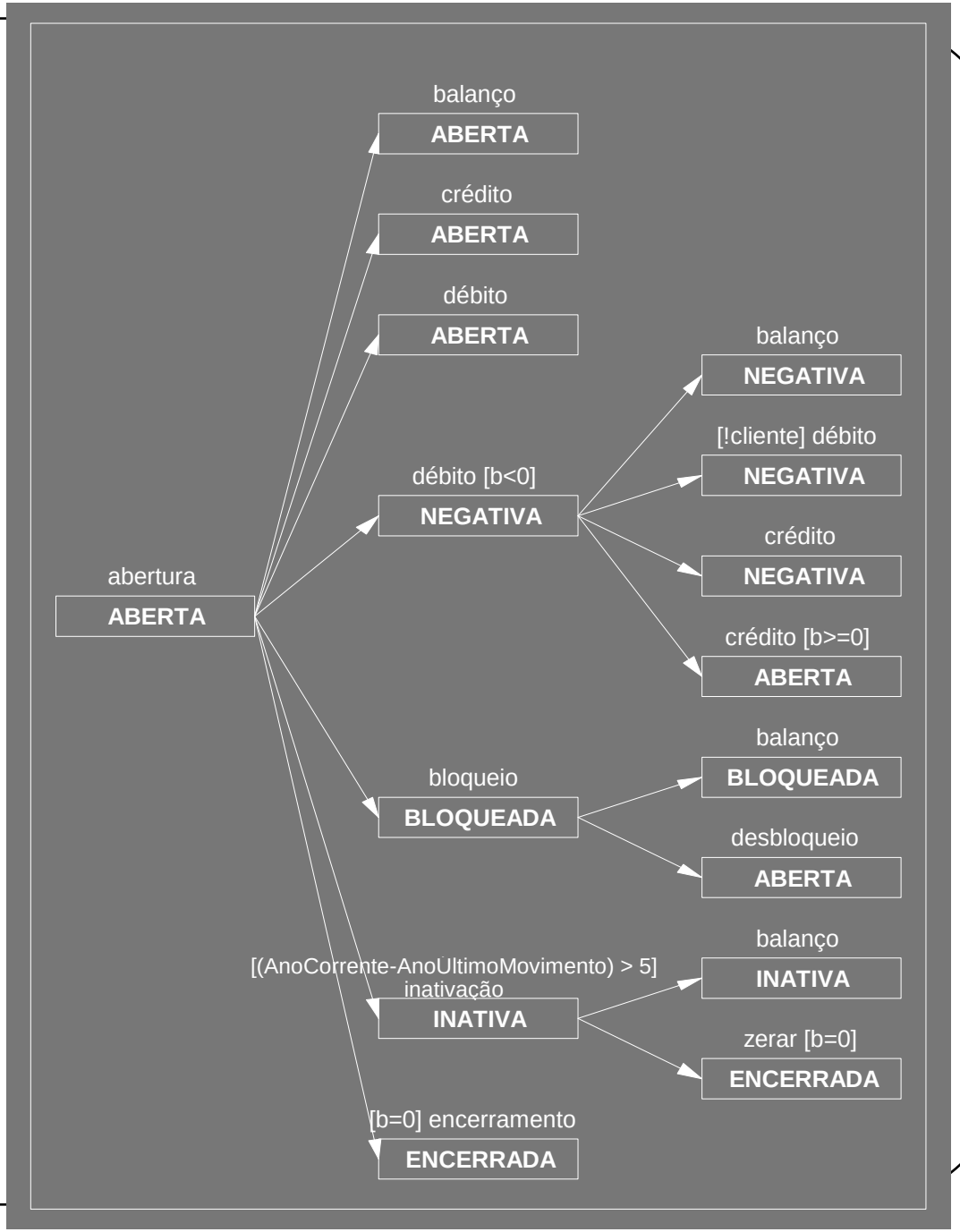
Classe Conta



Derivação de Seqüências de Teste

- Máquina de estados => árvore de transições
- Determinar seqüências de teste:
  - encadeamento de mensagens que modificam estado do objeto
  - raiz da árvore = estado inicial da máquina
  - para cada transição é desenhado um arco ligando estado origem a estado destino
- cada caminho a partir da raiz deriva no mínimo um caso de teste

Árvore de Transições



Árvore de Transições

- Tipos de erros detectados:
  - transições perdidas, incorretas, ações de saída incorretas ou estados incorretos
- completar plano de teste identificando parâmetros de métodos, e exceções
- execução dos testes:
  - criar objeto com o estado inicial (**classe driver**)
  - aplicar a seqüência de teste
  - comparar o estado resultante com o estado esperado

Critérios para Cobertura do Teste baseado em Estados [McGregor]

- Todos-Métodos: devem ser exercitados
  - não garante implementação de todos os estados necessários
- Todos-Estados: devem ser visitados
  - garante a implementação de todos os estados
- Todas-Transições
  - identifica estados e/ou métodos não implementados
- Todas-N-Transições: cobre combinações de transições
  - pode descobrir estados extras não especificados
- Todos-Caminhos: teste exaustivo, impraticável

Teste Incremental [Harrold]

- Objetivo:

  testar classes isoladamente, de forma incremental, reutilizando casos de teste, na hierarquia de herança

- abordagem top-down
  - teste inicia pela raiz e segue a árvore de herança
- teste das subclasses
  - teste de métodos novos
  - teste de métodos redefinidos e concretizados (abstratos)
  - integração de métodos redefinidos com novos

Tipos de Métodos

- **Novo**:
  - definido na subclasse
- **Herdado**:
  - definido apenas na superclasse
- **Redefinido**:
  - reimplementado na subclasse
- **Virtual ou Abstrato**:
  - interface apenas na superclasse
- **Virtual Redefinido ou Abstrato implementado**
  - implementado na subclasse

Teste das Subclasses

- História de teste da superclasse é transformada na história da subclasse.
- Métodos novos devem ser testados no contexto da subclasse
- Métodos herdados devem ser parcialmente retestados (**caso haja interação c/novos ou redefinidos**)
- Métodos redefinidos devem ser retestados
- Métodos abstratos implementados devem ser testados no contexto da subclasse

Teste da Superclasse

- Teste isolado de cada método
  - **provável uso de *drivers* e *stubs***.
- Histórias de teste são herdadas:

  $\{m_i,(TS_i,testar?),(TP_i,testar?)\}$

  TS - **conjunto de casos p/teste funcional**
  TP - **conjunto de casos p/teste estrutural**
- Valores de *testar?*: Y, N e P

  Y - teste deve ser executado
  N - teste não precisa ser executado
  P - teste deve ser parcialmente executado
- Casos de teste de unidade (classe) e integração (dentro da classe)

Teste de Métodos

- Abordagem funcional: divisão do domínio de entrada em partições de equivalência e análise dos limites
- Abordagem estrutural: executar cada um dos caminhos do método, considerando passar uma vez em cada laço

```
Class Shape
{     private:
        Point reference_point;
      public:
        void put_reference_point (Point);
        Point get_reference_point();
        void move_to(Point);
        void erase();
        void set_vertex3(Point);
virtual void draw();
virtual float area();
        Shape();
        Shape(Point);
}

Class Triangle: public Shape
{     private:
        Point vertex2;
        Point vertex3;
      public:
        Point get_vertex1();                // novo
        Point get_vertex2();                // novo
        Point get_vertex3();                // novo
        void set_vertex1(Point);            // novo
        void set_vertex2(Point);            // novo
        void set_vertex3(Point);            // novo
        void draw();                        // virtual-redefinido
        float area();                       // virtual-redefinido
        triangle();                         // novo
        triangle(Point,Point,Point);        // novo
}

Class EquiTriangle: public Triangle
{     public:
        float area();                               // redefinido
        equi_triangle(Point,Point,Point);           // novo
        equi_triangle();                            // novo
}
```

História de Teste para a classe Shape

MÉTODO	CASO DE TESTE FUNCIONAL	CASO DE TESTE ESTRUTURAL
teste de unidade		
put_reference_point	(TS $_1$,Y)	(TP $_1$,Y)
get_reference_point	(TS $_2$,Y)	(TP $_2$,Y)
move_to	(TS $_3$,Y)	(TP $_3$,Y)
erase	(TS $_4$,Y)	(TP $_4$,Y)
draw	(TS $_5$,Y)	----
area	(TS $_6$,Y)	(TP $_6$,Y)
shape	(TS $_7$,Y)	(TP $_7$,Y)
shape	(TS $_8$,Y)	(TP $_8$,Y)
teste de interação		
move_to	(TIS $_9$,Y)	(TIP $_9$,Y)
erase	(TIS $_{10}$,Y)	(TIP $_{10}$,Y)

História de teste para a classe Triangle

MÉTODO	CASO DE TESTE FUNCIONAL	CASO DE TESTE ESTRUTURAL
teste de unidade		
put_reference_point	(TS_1,N)	(TP_1,N)
get_reference_point	(TS_2,N)	(TP_2,N)
move_to	(TS_3,N)	(TP_3,N)
erase	(TS_4,N)	(TP_4,N)
draw	(TS_5,Y)	(TP_5,Y)
area	(TS_6,Y)	(TP_6,Y)
shape	(TS_7,N)	(TP_7,N)
shape	(TS_8,N)	(TP_8,N)
get_vertex1	(TS_{11},Y)	(TP_{11},Y)
get_vertex2	(TS_{12},Y)	(TP_{12},Y)
get_vertex3	(TS_{13},Y)	(TP_{13},Y)
set_vertex1	(TS_{14},Y)	(TP_{14},Y)
set_vertex2	(TS_{15},Y)	(TP_{15},Y)
set_vertex3	(TS_{16},Y)	(TP_{16},Y)
triangle	(TS_{17},Y)	(TP_{17},Y)
triangle	(TS_{18},Y)	(TP_{18},Y)
teste de interação		
move_to	($TIS"_9$,Y)	($TIP"_9$,Y)
erase	($TIS"_{10}$,Y)	($TIP"_{10}$,Y)
area	(TIS'_{19},Y)	(TIP'_{19},Y)
get_vertex1	(TIS'_{20},Y)	(TIP'_{20},Y)
set_vertex1	(TIS'_{21},Y)	(TIP'_{21},Y)

História de teste para a classe EquiTriangle

MÉTODO	CASO DE TESTE FUNCIONAL	CASO DE TESTE ESTRUTURAL
teste de unidade		
put_reference_point	(TS_1,N)	(TP_1,N)
get_reference_point	(TS_2,N)	(TP_2,N)
move_to	(TS_3,N)	(TP_3,N)
erase	(TS_4,N)	(TP_4,N)
draw	(TS_5,N)	(TP_5,N)
area	(TS_6,Y)	(TP_6,Y)
shape	(TS_7,N)	(TP_7,N)
shape	(TS_8,N)	(TP_8,N)
get_vertex1	(TS_{11},N)	(TP_{11},N)
get_vertex2	(TS_{12},N)	(TP_{12},N)
get_vertex3	(TS_{13},N)	(TP_{13},N)
put_vertex1	(TS_{14},N)	(TP_{14},N)
put_vertex2	(TS_{15},N)	(TP_{15},N)
put_vertex3	(TS_{16},N)	(TP_{16},N)
triangle	(TS_{17},N)	(TP_{17},N)
triangle	(TS_{18},N)	(TP_{18},N)
equi_triangle	(TS_{22},Y)	(TP_{22},Y)
equi_triangle	(TS_{23},Y)	(TP_{23},Y)
teste de interação		
move_to	(TIS"$_9$,P)	(TIP"$_9$,P)
erase	(TIS"$_{10}$,P)	(TIP"$_{10}$,P)
area	(TIS"$_{19}$,P)	(TIP"$_{19}$,P)

Considerações Finais

- Reutilização de classes exige alta confiabilidade
- Técnicas de teste OO:
  - algumas técnicas propostas
  - poucas ferramentas implementadas
  - métodos de geração de dados manuais
- maior interatividade ⇨ maior dificuldade em testar o sistema de software
- principal questão pendente: polimorfismo e ligação dinâmica ⇨ uso de reflexão computacional