




Laboratório de Engenharia de Software



Teste de Programas Orientados a Objetos

Arndt von Staa
Departamento de Informática
PUC-Rio
Junho 2015

Especificação



Laboratório de Engenharia de Software

- Objetivo desse módulo
 - discutir as características específicas do sistemas orientados a objetos e que afetam os testes.
- Justificativa
 - sistemas orientados a objetos têm uma série de características particulares. Entre elas: herança e forte preocupação com encapsulamento. Estas características adicionam dimensões específicas a estes tipos de sistemas quando se observa do ponto de vista do teste.
- Texto
 - Pezzè, M.; Young, M.; *Teste e Análise de Software*; Porto Alegre, RS: Bookman; 2008, capítulo 15

Jun 2015Arndt von Staa © LES/DI/PUC-Rio2

Características específicas



- Comportamento dependente do estado
 - **estado:**
 - o conjunto instantâneo de
 - valores atribuídos às variáveis membro de um objeto,
 - conhecimento quanto à qualidade desses valores
 - variáveis membro são, em última análise, variáveis globais para o conjunto de métodos do objeto
 - efeitos colaterais tornam o teste mais complexo
- Herança, polimorfismo, amarração dinâmica de funções
 - variáveis ponteiro para função
- Encapsulamento (aula teste estrutural 2)
- Classes abstratas (aula teste estrutural 3)
- Tratamento de exceções (aula teste estrutural 3)
- Concorrência, multi-threading (não é visto nesta disciplina)

Problemas específicos



- **Variabilidade**
 - métodos podem ser redefinidos → herança
 - polimorfismo → o comportamento nem sempre é o esperado
 - variações de assinaturas
 - » enganos ao programar e manutenção incorreta podem levar a assinaturas incompatíveis → não ocorre redefinição
 - comportamento polimórfico inesperado
 - » seja `class A { A::f'(int) }; class B:A { B::f''(int) };` se o objeto polimórfico foi criado com o construtor `A()`, então será executado `f'(int)`, possivelmente contrariando a expectativa do programador
 - como detectar?
 - » gerar casos de teste capazes de testar **todas as características polimórficas** → teste sem usar enchimentos ou mocks
 - » adicionar algum instrumento de *trace* seletivo (p.ex. *log*) capaz de informar o **corpo** do método executado. Podem-se criar verificadores de log para automatizar a inspeção
 - » realizar os testes e inspecionar os traces

Problemas específicos



- Variabilidade

- ponteiros para função (herança) podem receber referências não conhecidas quando do desenvolvimento inicial
 - ponteiro para função corresponde a uma família de funções, cada função atribuída será um dos membros dessa família
 - nenhum membro deve violar as regras que envolvem as assertivas da família ao estabelecer uma redefinição
 - AE família \Rightarrow AE membro; AE membro não restringe a AE família
 - AS membro \Rightarrow AS família; AS família não restringe a AS membro
 - propósito do membro deve ser um sub-propósito da família
 - » ex: pilha pode herdar de lista, mas jamais adicione, por herança, propriedades a uma pilha para que ela se torne uma lista
 - como detectar
 - especificar as assertivas em particular as da família e, se possível, automatizá-las,
 - conduzir uma inspeção nos membros para verificar a coerência de propósitos

Parêntesis: caracterizar *propósito*



- Propósito é um conceito recursivo
 - uma classe pode ter o propósito de implementar uma lista
 - provavelmente requererá outras classes
 - ex. *template* informando o tipo do elemento
 - » obs. um *template* é uma variável do tipo *tipo*, para quem tiver coragem: veja com é definido Algol68
 - ex. classe abstrata informando interface mínima do elemento
 - uma classe pode ter o propósito de manipular contas correntes
 - certamente interagirá com um conjunto grande de outras classes
 - o importante é que essa classe providencie **todas e somente as** operações de acesso a contas correntes:
 - criar, destruir, autorizar acesso, as diferentes formas de consultar, alterar o valor depositado e regras (ex. limite de saldo negativo) e nada mais do que isso
 - cálculo de juros não faz parte do propósito "**manipular** contas correntes"

Problemas específicos



- **Variabilidade**

- como assegurar que todas as redefinições requeridas, e somente elas, existem?
 - inspeção
 - verificação estática (compilador)
- como assegurar que todas as redefinições existentes foram testadas?
 - controle de cobertura
- no caso de evolução, como assegurar que todas as novas redefinições foram testadas?
 - controle de cobertura
- como assegurar que cada novo membro é estritamente coerente com a família?
 - ver slide anterior

Jun 2015

Arndt von Staa © LES/DI/PUC-Rio

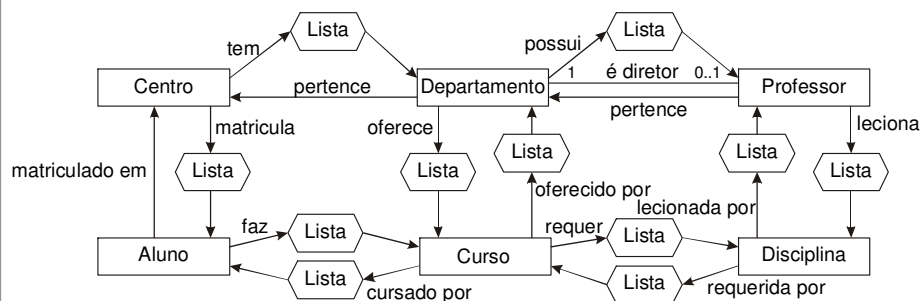
7

Problemas específicos



- **Circularidade**

- interdependência entre objetos
 - selecionar uma ordem de implementação que minimize o número de dublês a desenvolver
 - implementar usando os dublês e realizando um teste simples
 - à medida que os dublês forem substituídos, transformar os testes em testes mais rigorosos usando os módulos de produção



Jun 2015

Arndt von Staa © LES/DI/PUC-Rio

8

Problemas específicos



- **Espalhamento do código**
 - em OO é comum que o código de uma característica (*feature*), ou mesmo de uma funcionalidade, se espalhe por diversos métodos e possivelmente classes
 - dificulta entender o que o código faz e como o faz
 - é um caso de perda da **localidade de raciocínio**
 - algumas classes podem perder a característica de implementar **completa e exatamente** um único propósito relevante
 - um indicador de problema é a dificuldade de se dar um nome (frase curta) à classe, ou método, e que represente precisamente o seu propósito
 - examine os "design patterns" e tente dar um nome às classes que constituem a estrutura
 - como controlar? → usar **assertivas de saída** executáveis

Problemas específicos



- **Encapsulamento é inerentemente deficiente**
 - sintoma: propriedades encapsuladas são visíveis no texto do código tornado disponível para todos
 - faz parte da definição de uma classe tudo o que é protegido ou privado
 - é um problema inerente às linguagens usadas
 - ex. módulos de definição podem conter coisas do gênero:

```
int GetAlgo( ){ return algo ; }
```
 - isso torna possível a programadores cliente burlarem as regras de encapsulamento do código
 - Java nem sequer permite regras precisas quanto à separação de declaração e implementação
 - a falta de encapsulamento rígido torna mais difícil assegurar a corretude por construção
 - em particular ao manter

Problemas específicos



- **Completeza do teste**
 - tende a ser difícil **assegurar** que o teste é completo segundo algum critério de cobertura
 - **precisa-se medir** a cobertura para ter uma informação quanto à completeza do teste
 - mas a cobertura de estrutura – comandos, arestas, fragmentos de caminhos – não é suficiente para determinar se as funcionalidades estão corretas
 - como medir cobertura de características (*features*)?
 - todas as formas de uso corretos da característica
 - todas as formas de uso incorretos da característica
 - » isso é viável na prática?

Problemas específicos



- **Completeza do teste**
 - o teste deve levar em conta o **estado conceitual** do objeto
 - exemplos de estados conceituais
 - pilha vazia, pilha não vazia nem cheia, pilha cheia no caso de uma pilha de tamanho pré-fixado
 - disciplina selecionada ao criar uma solicitação de matrícula
 - disciplina válida
 - o valor continua o mesmo que o de uma disciplina selecionada, mas sabemos mais a respeito da sua qualidade,
 - ex. para cada disciplina selecionada ao matricular, será válida sse:
 - » existe no catálogo
 - » é oferecida
 - » o horário não possui superposição com o de outras disciplinas selecionadas
 - » o aluno tem os pré-requisitos necessários
 - » o total de créditos selecionado pelo aluno não ultrapassa o máximo de créditos que pode cursar no semestre

Problemas específicos



- **Completeza do teste**
 - sabemos que software em uso conterà defeitos remanescentes, o que fazer para impedir lesões?
 - uma possível resposta: manter assertivas suficientemente rigorosas em pontos estratégicos do código
 - observam os erros, reportam falhas
 - não necessariamente impedem pequenos danos (lesões), mas podem impedir grandes danos (ausência de lesões vultosas)
 - observada uma falha: adicionar caso de teste que a evidencie, depois corrigir
 - estabelecer níveis de granularidade para as assertivas,
 - à medida que o desenvolvimento progride, as assertivas de granularidade fina são desativadas (compilação condicional)
 - hipótese: o desenvolvimento iterativo usando módulos já aprovados tende a exercitar esses módulos de forma diferente dos testes.

Lesão: dano provocado por erros não observados


Recomendações



- Em OO é fundamental projetar-se e implementar-se visando **testabilidade**
 - desenvolvedor e testador precisam interagir durante o desenvolvimento e o projeto dos testes
 - em especial durante o teste de unidade, de componentes e de arcabouços (*frameworks*)
 - nesta etapa usualmente são a mesma pessoa

Laboratório de Engenharia de Software

Recomendações




- Procure utilizar
 - desenvolvimento dirigido por testes
 - teste automatizado de unidades
 - integração contínua
 - especificação usando contratos (assertivas) executáveis
 - entrada, saída, invariantes da classe e invariantes estruturais
 - vários ambientes de desenvolvimento oferecem bibliotecas de apoio a contratos
 - C# e outros .NET -> Code contracts
 - Java -> Java modeler + ESC/Java2
 - existem pacotes para apoio a contratos, incluindo verificação estática deles
 - desenvolvimento incremental das classes, componentes e arcabouços

Jun 2015
Arndt von Staa © LES/DI/PUC-Rio
15

Laboratório de Engenharia de Software

Recomendações



- Assegure elevada **detectabilidade** e **diagnosticabilidade**
 - utilize sempre assertivas executáveis de entrada, mesmo nas funções privadas
 - **contratos**: são assertivas que envolvem **somente elementos da interface** e propriedades conceituais manipuláveis pelos módulos cliente
 - em muitos casos assertivas pontuais são insuficientes → estruturas de dados complexas
 - **assertivas abrangentes**: assertivas envolvendo a interface conceitual dos métodos e as estruturas de dados usadas
 - mesmo se envolverem vários objetos de diferentes classes
 - o que fazer depois de testado?
 - deixe-as no código até pelo menos a etapa de teste de sistema
 - se não incomodarem (desempenho), deixe-as no código de produção
 - assertivas estruturais podem ser transformadas em um thread
 - » esforço para isso pode ser grande

Jun 2015
Arndt von Staa © LES/DI/PUC-Rio
16

Características que afetam testes



- Tipos de classes
 - **primitivo**
 - não se relaciona com outra classe
 - **coleção** (*container, collection*)
 - registra objetos, mas não opera sobre eles
 - **colaborador**
 - relaciona com outras classes e opera sobre os objetos relacionados
 - classe cliente e classe servidora (colaboradora)


Características que afetam testes



- Como desenvolver sistemas incrementalmente?
 - desenvolvimento e teste classe a classe
 - integração somente de classes aprovadas
 - **primitivas** → não requerem colaboradores
 - **dependentes** → dependem de colaboradores
 - podem necessitar de classe dublê – enchimento ou imitação
 - Aula: Teste Estrutural 2
 - procure implementar segundo uma sequência que minimize a necessidade de dublês

Laboratório de Engenharia de Software

Teste de classes primitivas




- Classes primitivas sem herdeiros
 - podem ser testadas de forma convencional
- Classes primitivas com herdeiros
 - teste a superclasse de forma isolada
 - se contiver métodos abstratos: crie um herdeiro enchimento (dublê) para viabilizar a instanciação e a verificação dos parâmetros
 - no módulo de teste específico desenvolva uma fábrica de objetos capaz de considerar cada um dos herdeiros
 - testar todos os herdeiros conhecidos
 - considerar os métodos redefinidos e os adicionados em cada um dos herdeiros
 - procure automatizar as assertivas da família

Jun 2015
Arndt von Staa © LES/DI/PUC-Rio
19

Laboratório de Engenharia de Software

Teste de herança




- Sejam duas classes *A* e *B* em que *B* é herdeira de *A*
 - elemento **nov**o: elemento definido em *B* mas não em *A*
 - elemento **herdado**: elemento definido em *A* mas não em *B*
 - método **redefinido**: método definido em *A* e *B* e ambos possuem a mesma lista de parâmetros
 - mesmo nome em *B* com parâmetros diferentes de *A* não é herança, é sobrecarga (*overloading*) de nome
 - método **virtual novo**: método virtual especificado em *B*
 - método **virtual herdado**: método virtual especificado em *A* mas não implementado em *B*
 - método **virtual redefinido**: método virtual especificado em *A* e implementado em *B*,
 - com a mesma assinatura (tipos de parâmetros)

Jun 2015
Arndt von Staa © LES/DI/PUC-Rio
20

Laboratório de Engenharia de Software

Teste de coleções




- Teste coleções de forma similar a classes primitivas
 - crie objetos e os insira na coleção
 - se for o caso, verifique se as ordenações ou repetições estão corretas
 - obtenha objetos específicos e verifique se o conteúdo dos objetos são os esperados
 - no caso de inserção de cópias (*clones*), verifique se os objetos são copiados corretamente
 - as referências contidas no objeto são copiadas corretamente?
 - clonagem do objeto ou duplicação da estrutura referenciada?
 - verifique se a coleção é capaz de crescer além do número máximo estimado de elementos
 - se o limite de crescimento for desconhecido, procure uma ferramenta que permita simular limitação de memória
 - verifique se o tratamento da destruição da coleção trata de forma correta todos os membros da coleção

Jun 2015
Arndt von Staa © LES/DI/PUC-Rio
21

Laboratório de Engenharia de Software

Teste de colaborações




- Caso uma ou mais classes colaboradoras ainda não esteja implementada
 - crie módulos dublê:
 - enchimento (*stub*)
 - imitação (*mock*)
 - assegure que as interfaces do dublê sejam exatamente iguais às da futura classe de produção
 - assegure que os casos de teste sejam coerentes com o serviço prestado pelos dublês
 - teste o conjunto como uma classe primitiva
 - considere as diversas formas de herança

Jun 2015
Arndt von Staa © LES/DI/PUC-Rio
22

Laboratório de Engenharia de Software

Teste de colaborações




- À medida que os dublês forem sendo substituídos pelos módulos de produção
 - reformule os testes de modo que operem corretamente com os módulos de produção
 - pode ser mais conveniente realizar a reformulação dos testes em bloco quando todos (ou um conjunto) de dublês tiver sido substituído
 - isso facilitará mais adiante a manutenção e o desenvolvimento incremental
 - torne os testes mais abrangentes (rigorosos) ao eliminar os dublês

Jun 2015
Arndt von Staa © LES/DI/PUC-Rio
23

Laboratório de Engenharia de Software

Teste de colaborações



- No caso de um artefato possuindo n módulos interdependentes pode-se seguir o caminho:
 - criar um conjunto de n módulos específicos de teste
 - durante o desenvolvimento do componente: teste sem requerer que os objetos servidores estejam presentes – use objetos dublê
 - gerar os correspondentes programas de teste
 - efetuar os testes dos n módulos
 - através de uma espécie de “make de makes” pode-se conduzir o teste de regressão
 - para um exemplo ver no arcabouço de teste Talisman:
 - `tools\programs\RunTestSuite.lua`
 - `test\testcase\TestFramework.suite`
 - existem várias ferramentas no mercado
 - quando o artefato estiver completo: reformular o teste de modo a usar os módulos de produção
 - mantenha mocks para servidores específicos, ex. base de dados

Jun 2015
Arndt von Staa © LES/DI/PUC-Rio
24

Laboratório de Engenharia de Software

Teste dependente de estado

- Um **protocolo** define as sequências permitidas de chamadas dos métodos de um objeto, considerando o estado semântico deste objeto
 - **estado semântico** – propriedade conhecida ao cliente do objeto ou da estrutura, independentemente da forma adotada ao implementar

```

graph LR
    Start(( )) -- "Criar pilha" --> Vazia[Pilha vazia]
    Vazia -- "Desempilhar" --> Excepcao1[Exceção]
    Vazia -- "Empilhar" --> NaoVazia[Pilha não vazia]
    NaoVazia -- "Empilhar" --> Cheia[Pilha cheia]
    Cheia -- "Empilhar" --> Excepcao2[Exceção]
    Cheia -- "Desempilhar" --> NaoVazia
    NaoVazia -- "Desempilhar" --> Vazia
    NaoVazia -- "{ficou cheia}" --> End1(( ))
    NaoVazia -- "{else}" --> End2(( ))
    NaoVazia -- "{ficou vazia}" --> End3(( ))
    NaoVazia -- "{else}" --> End4(( ))
  
```

Jun 2015
Arndt von Staa © LES/DI/PUC-Rio
25

Laboratório de Engenharia de Software


Teste dependente de estado

- Protocolos podem ser representados por máquinas de estado
 - transições são rotuladas com o nome dos métodos
 - condições podem aparecer como consequencia de realização de um método
 - muitas vezes usam-se redes de Petri: permitem verificar propriedades de execução paralela
- Teste o protocolo
 - tabela de decisão estado a estado
 - Aula Máquinas de Estados
 - ou todos os caminhos segundo as regras de teste de caminhos
 - Aula Teste Estrutural 2

Jun 2015
Arndt von Staa © LES/DI/PUC-Rio
26

Laboratório de Engenharia de Software

Teste dependente de estado



- Condições do método `obterEndereçoReal(endereçoVirtual)`
 - página já está em um portador
 - existe portador vazio
 - existe portador utilizável (contém outra página, mas não está "pinado")
 - portador utilizável está marcado alterado
- Condições de saída Sem considerar erros de uso
 - retornou o portador que contém a página
 - o portador indica o endereço virtual da página que contém
 - o portador está no início da lista LRU
 - o portador está na lista de colisão correta
 - selecionou o portador vazio mais antigo ou o portador utilizável mais antigo
 - se alterada, gravou a página virtual velha contida no portador utilizável
 - leu a página virtual nova para o portador escolhido
 - exceção caso não consiga encontrar portador a ser escolhido


Jun 2015

Arndt von Staa © LES/DI/PUC-Rio

27

Laboratório de Engenharia de Software

Arcabouços (*framework*)

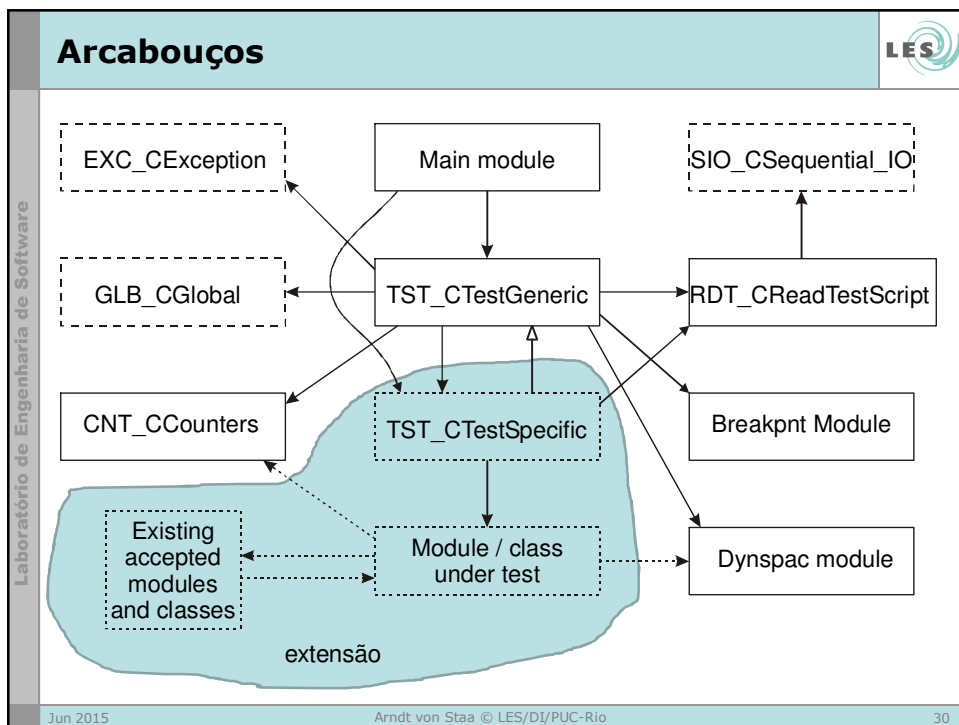
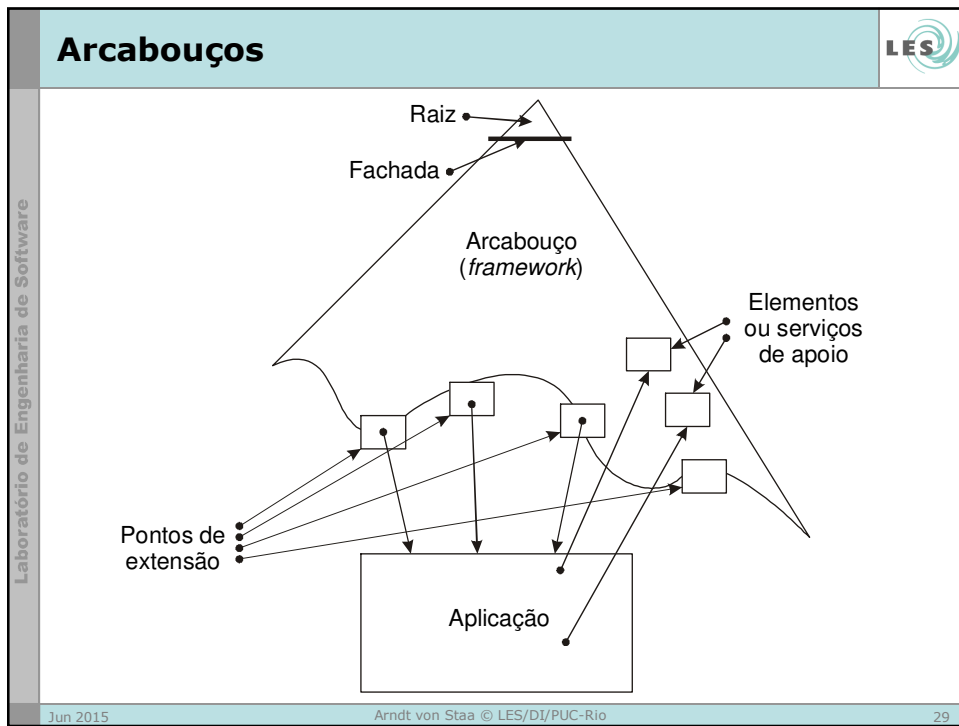


- Um arcabouço é uma **solução incompleta** que precisa ser completada para resolver um problema específico
- Arcabouços possuem **pontos de extensão** (*hot spots*)
 - em geral classes abstratas
- Arcabouços devem implementar **um único conceito** bem definido
 - ex. apoio ao teste de módulos
 - ex. apoio à implementação de interfaces gráficas (GUI)
- Arcabouços podem ser as **origens** (*main*) das aplicações
 - ex. JUnit
- Arcabouços podem ter uma **fachada** (ou similar) como "origem"
 - ex. arcabouço Talisman para teste C++

Jun 2015


Arndt von Staa © LES/DI/PUC-Rio

28



Laboratório de Engenharia de Software

Arcabouços




- Como testar?
 - ao desenvolver:
 - teste módulo a módulo incremental convencional
 - uma vez desenvolvido:
 - criar uma aplicação de imitação (*mock*)
 - controla a sequência de chamadas – protocolo
 - » ex. os objetos da aplicação são fabricados antes de serem utilizados?
 - controla a recepção de dados
 - simula comportamento errôneo
 - se possível criar uma coletânea de programas de teste em que cada classe do arcabouço é testada no contexto do arcabouço
 - teste de regressão

Jun 2015
Arndt von Staa © LES/DI/PUC-Rio
31

Laboratório de Engenharia de Software

Arcabouços



- Herança e protocolos
 - protocolos e contratos **devem estar especificados** no arcabouço
 - devem ser seguidos fielmente pela aplicação instanciada
 - o teste do protocolo na aplicação imitação verifica a correteude do arcabouço para supostamente qualquer instanciação
 - aplicações instanciadas precisam testar os protocolos para verificar se a aplicação está em conformidade com eles
 - estados podem ter sido decompostos ao instanciar

Jun 2015
Arndt von Staa © LES/DI/PUC-Rio
32

Componentes



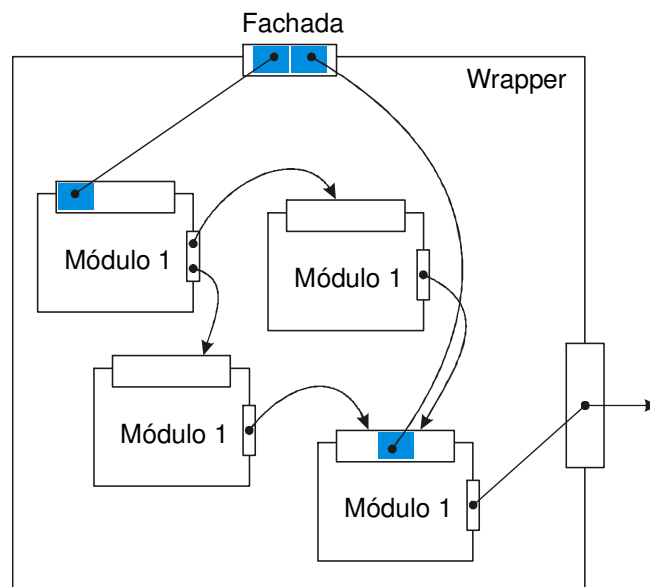
- Componentes são soluções (quase) **completas** para um serviço específico e **bem delimitado**
 - são compostos por um ou mais módulos (classes)
 - são apresentados tipicamente na forma de DLLs , LIBs, JARs , objetos Corba, objetos Com, JavaBeans
 - componentes empacotam e encapsulam a implementação
 - a interface deve disponibilizar somente o que interessa para potenciais clientes (módulos, componentes)
 - implica a seleção do que deve ser visível e encapsulamento do restante

Jun 2015

Arndt von Staa © LES/DI/PUC-Rio

33

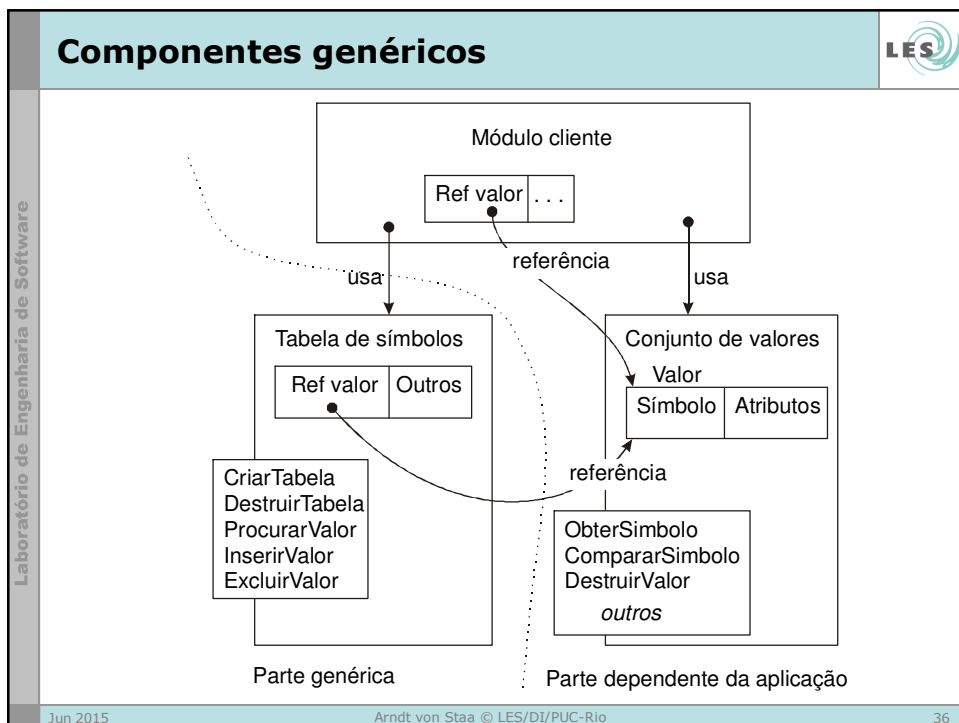
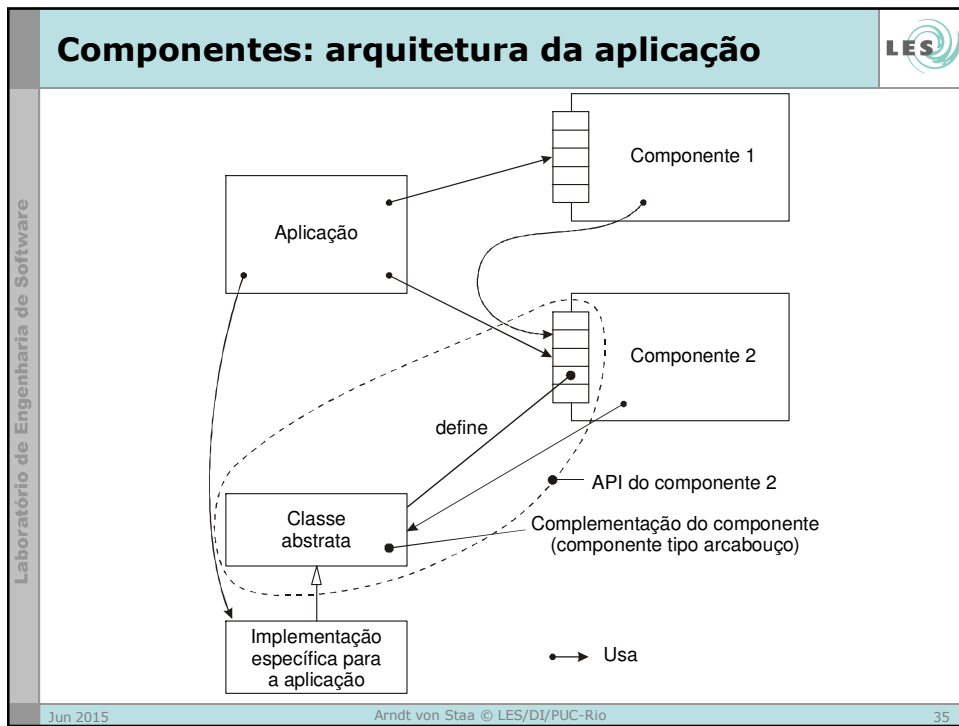
Componentes: organização fachada



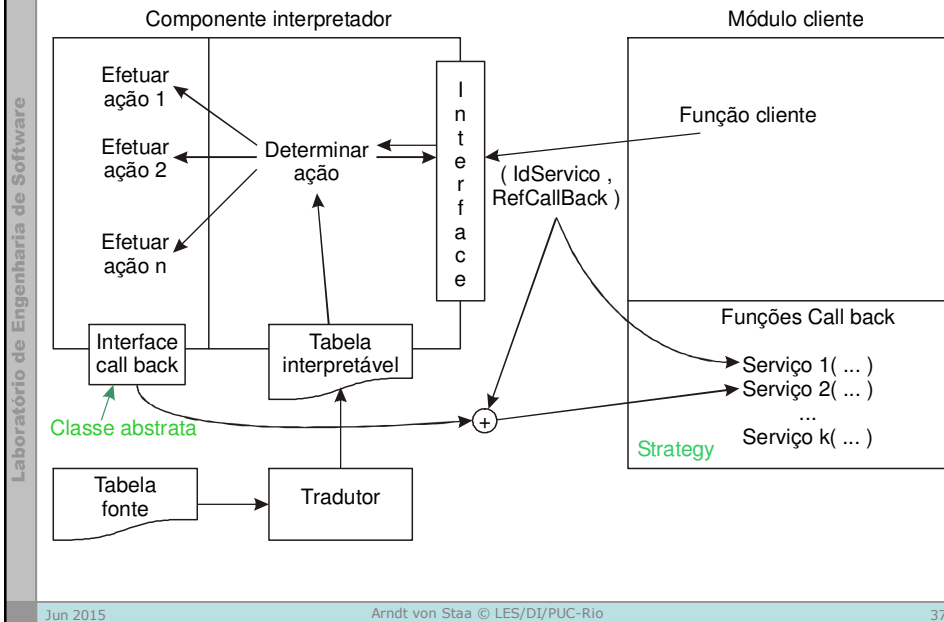
Jun 2015

Arndt von Staa © LES/DI/PUC-Rio

34



Componentes: interpretador (meta-componente)

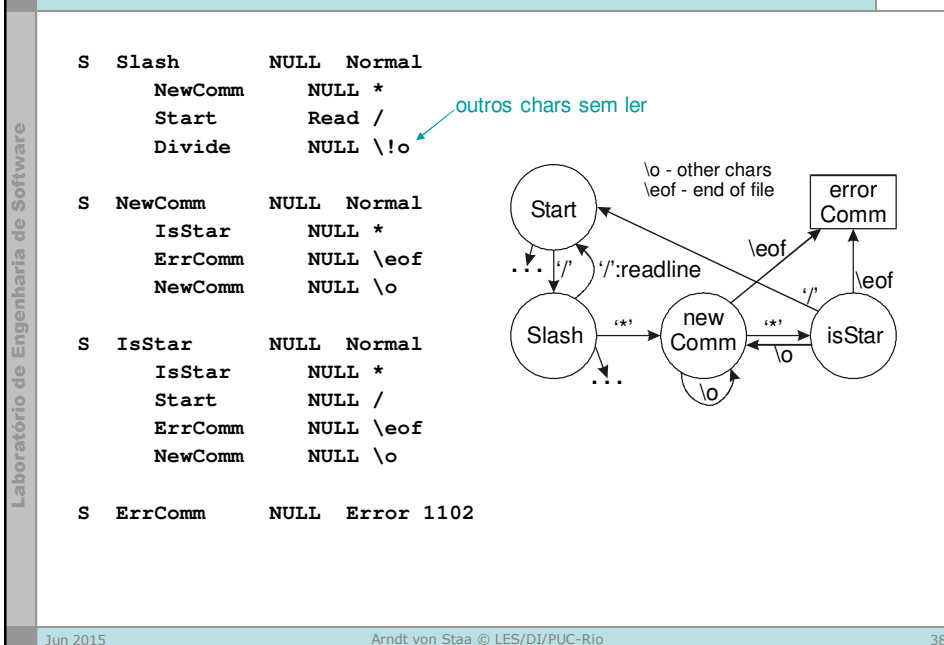


Jun 2015

Arndt von Staa © LES/DI/PUC-Rio

37

Autômato – interpretador comentários C++



Jun 2015

Arndt von Staa © LES/DI/PUC-Rio

38

Componentes como testar



- Teste incremental do componente: convencional
- Teste da interface com um dublê *driver*
 - imita a forma de uso do componente
 - link dinâmico
 - plug and play – plug-in
 - acesso remoto
 - teste de sincronização se o componente admite multi-programação ou multi-processamento
 - testa uso incorreto da interface → verificar os controles dos contratos e assertivas executáveis
 - mede cobertura da interface
- Teste com aplicação(ões) cliente típica(s)
 - mede cobertura da interface

Referências bibliográficas



- Delamaro, M.E.; Maldonado, J.C.; Jino, M.; *Introdução ao Teste de Software*; Rio de Janeiro, RJ: Elsevier / Campus; 2007
- McGregor, J.D.; Sykes, D.A.; *A Practical Guide to Testing Object-Oriented Software*; Reading, Massachusetts: Addison-Wesley; 2001
- Pezzè, M.; Young, M.; *Teste e Análise de Software*; Porto Alegre, RS: Bookman; 2008

Laboratório de Engenharia de Software

LES

FIM

Jun 2015

Arndt von Staa © LES/DI/PUC-Rio

41