

Desenvolvimento de Sistemas Confiáveis centrado na Arquitetura usando Tratamento de Exceções

Profa. Dra. Cecília Mary Fischer Rubira

Instituto de Computação - UNICAMP

2013



Roteiro da apresentação

- Introdução à engenharia de software
- Princípios de tolerância a falhas
- Tratamento de exceções
- Arquitetura de software
- Desenvolvimento centrado na arquitetura usando tratamento de exceções
- Tratamento de exceções concorrentes (cooperativas)
- Arquiteturas contemporâneas com tratamento de exceções
- Considerações finais



Parte 1

Introdução à Engenharia de Software



Engenharia de Software (I)

- Engenharia de software é a disciplina que abrange práticas, métodos e ferramentas para desenvolver e entregar sistemas de software que sejam úteis.
- É uma engenharia recente quando comparada com outras...



Engenharia de Software (II)

Diferente das engenharias tradicionais:

- Complexidade de software
- Fator humano (criatividade)
- Volatilidade de requisitos
- Invisibilidade
- Propagação de defeitos
- Comunicação





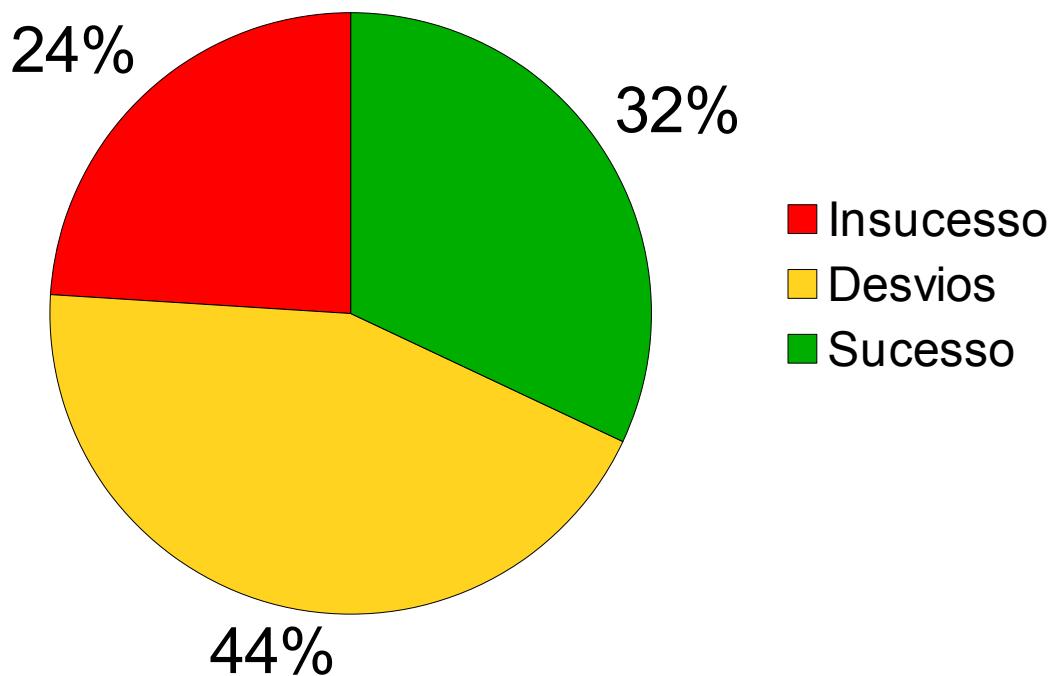
Em 1968...

- *First Software Engineering Conference* da OTAN, Garmisch-Alemanha, cunhou o termo **crise de software**.
- Problemas com o desenvolvimento de software:
 - qualidade baixa dos programas;
 - dificuldade de estimar prazos;
 - custo alto de manutenção;
 - duplicação de esforços.



... E a Situação Atual?

Dados sobre projetos de software



LEGENDA

Insucesso: projetos cancelados ou entregues e nunca usados

Desvios: projetos com custo ou prazo maior que o esperado, ou com quantidade menor de requisitos implementado.

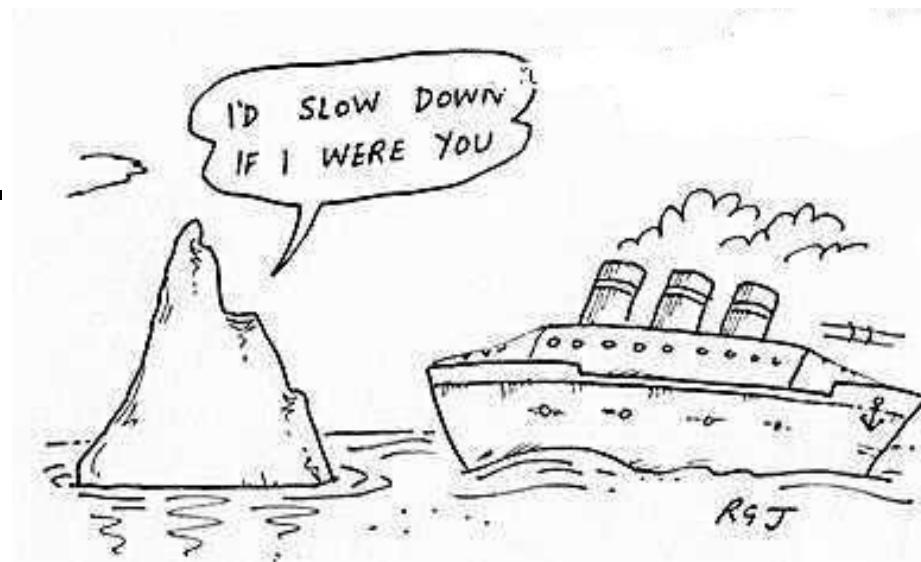
Sucesso: o projeto atendeu todos os requisitos no tempo e custo previamente estimados.

Fonte: Standish report, Chaos 2009



No caminho do insucesso

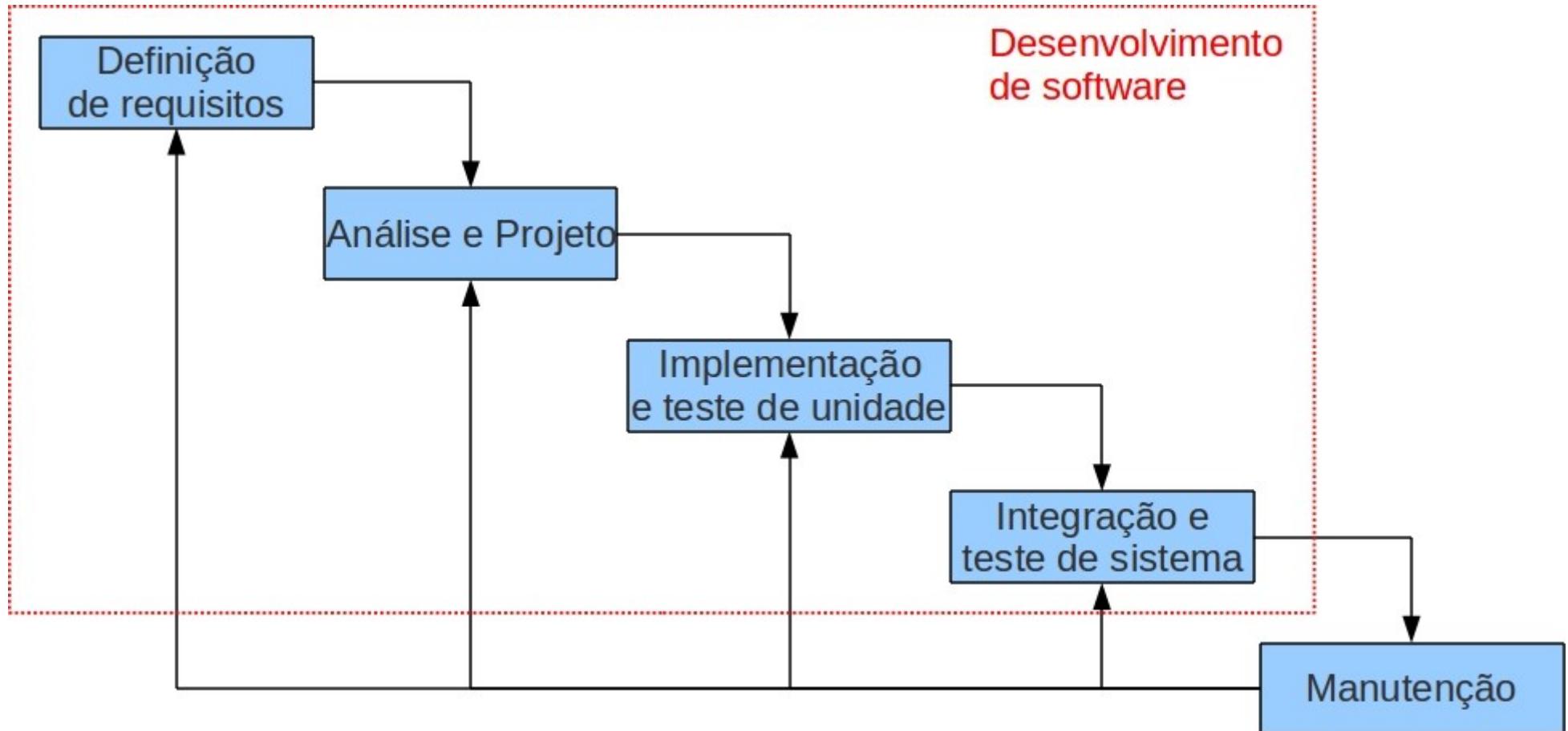
- Expectativas e prazos irrealistas.
- Requisitos incompletos e voláteis.
- Pouco envolvimento de usuários para especificar sistemas.
- Problemas com recursos.
- Falta de apoio à gestão.
- Falta de planejamento.



THE TIP OF THE ICEBERG



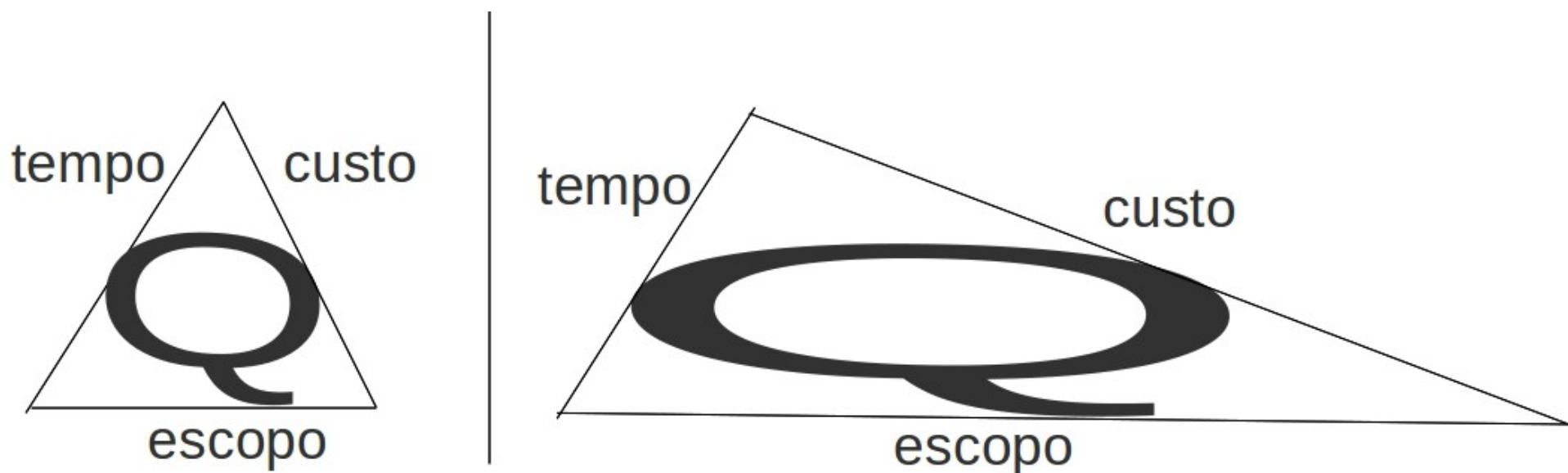
Desenvolvimento de Software





Gerenciamento de Projetos

- Projeto de Software é um esforço temporário com a finalidade de criar um produto único.
- 3 mosqueteiros (que são 4 com D'artagnan: qualidade)





O que é qualidade então?

- Dicionário Aurélio:
 - Propriedade, atributo ou condição das coisas ou das pessoas que à distingue das outras e lhes determina a natureza.
 - Superioridade, excelência de algo.



Garantia de qualidade: duas partes

- O **produto** entregue:
 - Livre de defeitos (*failures*)
 - Deve resolver o problema - Requisitos
 - Atividades: teste, revisões, inspeções, métodos formais, etc.
- O **processo** usado para construir o produto:
 - Para limitar a variação humana
 - Como desenvolver um produto de qualidade de forma consistente?
 - Atividades: auditorias, inspeções, CMM, etc.



Como combater o insucesso?

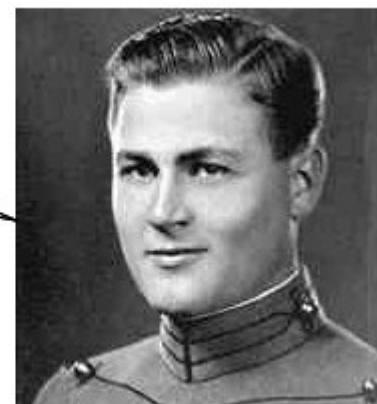
- Apostar na Qualidade
 - Processo de desenvolvimento de software
 - Práticas de engenharia
 - Normas
- Exigência desde o início
 - A qualidade não pode ser injetada no final
- Mahatmma Gandhi: "*Não existe um caminho para a felicidade. A felicidade é o caminho.*"
qualidade qualidade



Complexidade de Software (I)

- F. Brooks: "Entidades de software são mais complexas (...) do que qualquer outra coisa produzida pelo homem."
- A **complexidade** de software é uma propriedade essencial, não acidental.
- Dificuldades essenciais são inerentes à natureza do software e acidentais apresentam-se hoje na sua produção, mas não são inerentes

Se alguma coisa pode dar errado, com certeza dará.



Edward
Murphy



Complexidade de Software (II)

- Sistemas de software têm um número de estados em ordem de grandeza muito maior que computadores digitais.
- Traz dificuldades:
 - comunicação entre membros da equipe.
 - para enumerar todos os estados
 - para evoluir
- Leva a deficiências no produto.
- Leva ao aumento do custo e atrasos no cronograma.



Exemplos de Sistemas Complexos



- Complexidade de alguns sistemas (em milhões de linhas de código (MLOC)) :
 - O sistema aviônico do F-22 Raptor ~1,7 MLOC.
 - O sistema aviônico Boeing 787 Dreamliner ~6,5 MLOC.
 - Os sistemas de navegação e radio do Mercedes-Benz S-class têm ~20,0 MLOC.
- Tendência de aumento:
 - Carros que analisam o ambiente (e.g. baliza, distância entre carros, etc)

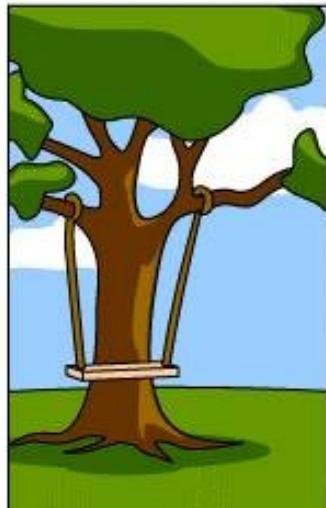




Diferentes pontos de vista



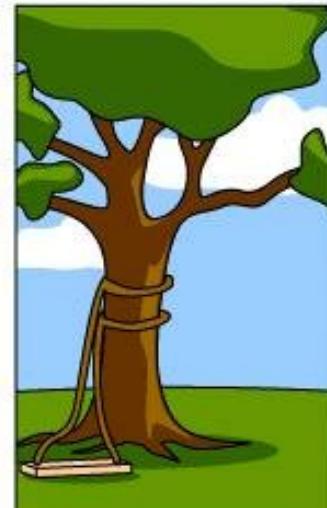
How the customer explained it



How the Project Leader
understood it



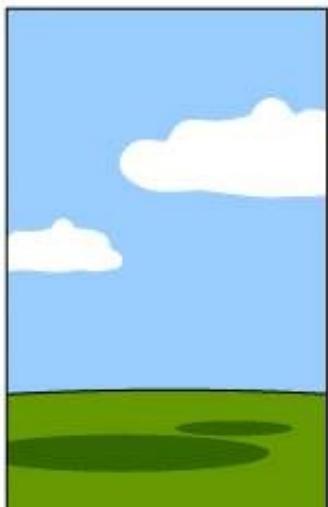
How the Analyst designed it



How the Programmer wrote it



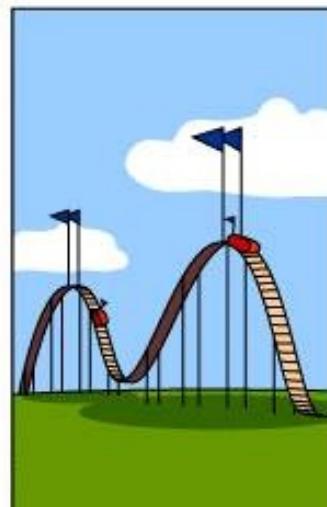
How the Business Consultant
described it



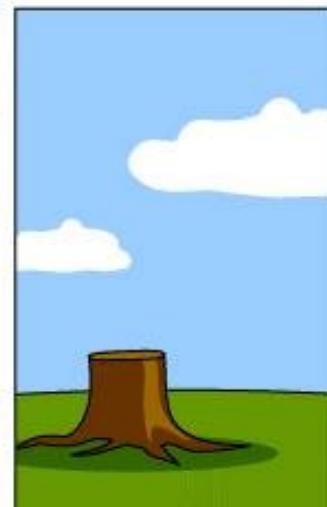
How the project was
documented



What operations installed



How the customer was billed



How it was supported



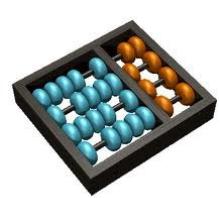
What the customer really
needed



Processo de Criação/Invenção

F. Brooks

- 1) Concepção do software
- 2) Implementação do software
- 3) Interação dos usuários finais com o software
 - Só então a sua criação foi finalizada
 - É fácil construir um software complexo sem defeitos?



Prejuízos causados por software

R. Charette. Why software fails. IEEE Spectrum, 2005.

YEAR	COMPANY	OUTCOME (COSTS IN US \$)
2005	Hudson Bay Co. [Canada]	Problems with inventory system contribute to \$33.3 million* loss.
2004–05	UK Inland Revenue	Software errors contribute to \$3.45 billion* tax-credit overpayment.
2004	Avis Europe PLC [UK]	Enterprise resource planning (ERP) system canceled after \$54.5 million† is spent.
2004	Ford Motor Co.	Purchasing system abandoned after deployment costing approximately \$400 million.
2004	J Sainsbury PLC [UK]	Supply-chain management system abandoned after deployment costing \$527 million.†
2004	Hewlett-Packard Co.	Problems with ERP system contribute to \$160 million loss.
2003–04	AT&T Wireless	Customer relations management (CRM) upgrade problems lead to revenue loss of \$100 million.
2002	McDonald's Corp.	The Innovate information-purchasing system canceled after \$170 million is spent.
2002	Sydney Water Corp. [Australia]	Billing system canceled after \$33.2 million† is spent.
2002	CIGNA Corp.	Problems with CRM system contribute to \$445 million loss.
2001	Nike Inc.	Problems with supply-chain management system contribute to \$100 million loss.
2001	Kmart Corp.	Supply-chain management system canceled after \$130 million is spent.
2000	Washington, D.C.	City payroll system abandoned after deployment costing \$25 million.
1999	United Way	Administrative processing system canceled after \$12 million is spent.
1999	State of Mississippi	Tax system canceled after \$11.2 million is spent; state receives \$185 million damages.
1999	Hershey Foods Corp.	Problems with ERP system contribute to \$151 million loss.
1998	Snap-on Inc.	Problems with order-entry system contribute to revenue loss of \$50 million.
1997	U.S. Internal Revenue Service	Tax modernization effort canceled after \$4 billion is spent.
1997	State of Washington	Department of Motor Vehicle (DMV) system canceled after \$40 million is spent.
1997	Oxford Health Plans Inc.	Billing and claims system problems contribute to quarterly loss; stock plummets, leading to \$3.4 billion loss in corporate value.
1996	Arianespace [France]	Software specification and design errors cause \$350 million Ariane 5 rocket to explode.
1996	FoxMeyer Drug Co.	\$40 million ERP system abandoned after deployment, forcing company into bankruptcy.
1995	Toronto Stock Exchange [Canada]	Electronic trading system canceled after \$25.5 million** is spent.
1994	U.S. Federal Aviation Administration	Advanced Automation System canceled after \$2.6 billion is spent.
1994	State of California	DMV system canceled after \$44 million is spent.
1994	Chemical Bank	Software error causes a total of \$15 million to be deducted from 100 000 customer accounts.
1993	London Stock Exchange [UK]	Taurus stock settlement system canceled after \$600 million** is spent.
1993	Allstate Insurance Co.	Office automation system abandoned after deployment, costing \$130 million.
1993	London Ambulance Service [UK]	Dispatch system canceled in 1990 at \$11.25 million**; second attempt abandoned after deployment, costing \$15 million.**
1993	Greyhound Lines Inc.	Bus reservation system crashes repeatedly upon introduction, contributing to revenue loss of \$61 million.
1992	Budget Rent-A-Car, Hilton Hotels, Marriott International, and AMR [American Airlines]	Travel reservation system canceled after \$165 million is spent.



Parte 2: Princípios de Tolerância a Falhas



Espectro de Prejuízos

- Prejuízos do mal-funcionamento vão desde simples inconveniências até perda de vidas e dinheiro.
- Sistemas de software devem ser confiáveis (*dependable*)

inconveniências



perda de vidas e/ou muito dinheiro



espectro



Software Dependability

- *Dependability* é a propriedade de um sistema que possibilita a seus usuários, justificadamente, "dependerem" dos serviços que ele oferece. [colocar a máquina de café do IC]



- A máquina de café da copa é dependable?
- A comunidade do IC pode justificadamente depender do seu serviço de entrega de café?
- Esse sistema já te causou prejuízos?



Dimensões de *Dependability*

- 1) *Availability* é a capacidade do sistema em oferecer os serviços quando requeridos;
- 2) *Reliability* é a capacidade do sistema em oferecer os serviços como especificado;
- 3) *Safety* é a capacidade do sistema operar sem defeitos catastróficos;
- 4) *Security* é a capacidade do sistema proteger-se contra intrusão accidental ou deliberada.

Atributo de qualidade (ou requisito não-funcional) é uma restrição nos serviços ou funções oferecidas pelo sistema.



Sistemas de Software Confiáveis

- Usuários de software esperam que todos os sistemas sejam confiáveis.
- Para aplicações não-críticas eles podem aceitar alguns **defeitos** do sistema.
- Aplicações críticas exigem requisitos de alta confiabilidade.
- Técnicas especiais de engenharia de software são usadas para prover esse atributo de qualidade



Falha, Erro e Defeito (I)

- **Defeito (*failure*)** é quando o sistema deixa de oferecer as funcionalidades previstas, ou o faz em desacordo com sua especificação.
- Exemplo: existe um defeito no caixa eletrônico quando não é permitido a um cliente fazer um saque nas condições de seu contrato.

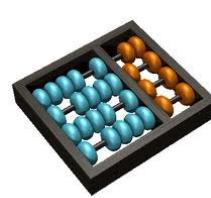




Falha, Erro e Defeito (II)

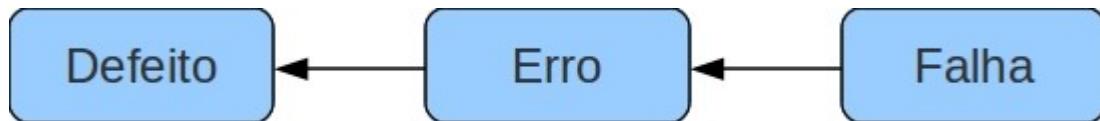
- Um **erro** (*error*) é uma parte do estado interno do componente, que sob determinadas entradas provoca um defeito.
- Exemplo: o saque foi negado em função de um valor incorreto do saldo da sua conta.





Falha, Erro e Defeito (III)

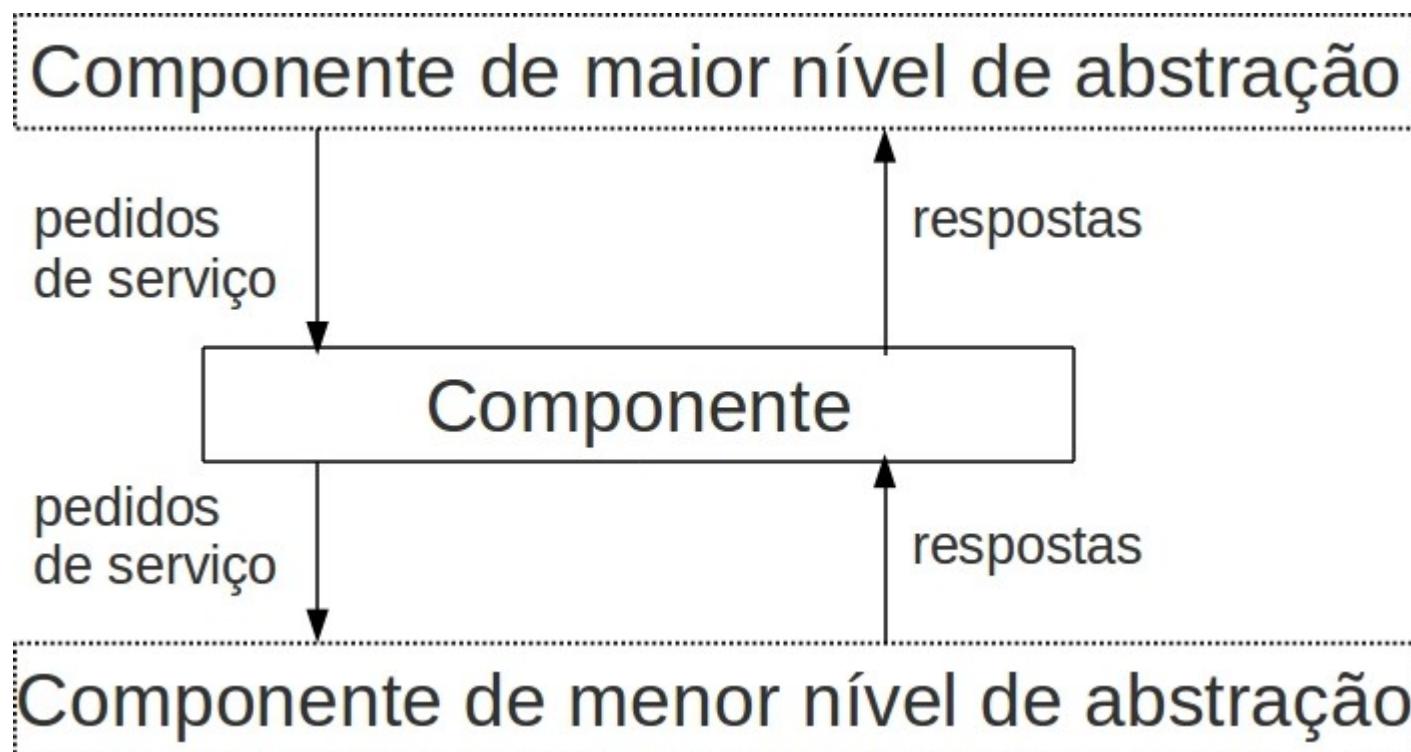
- Uma **falha** (*fault*) é um evento ou sequência de eventos que propicia o surgimento de um erro.
- Exemplo: um valor incorreto do saldo foi causado por uma falha na transmissão entre o computador do banco e o caixa eletrônico.





Estruturação de Sistemas

- É essencial para controlar a complexidade de um sistema de software.
- Um **sistema** consiste em um conjunto de componentes que interagem sob o controle de um projeto (*design*).





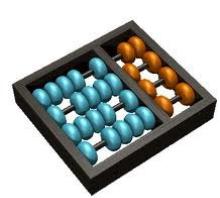
Tipos de Falhas

- **Falha de componente** (*component fault*)
 - erro no estado interno do componente
 - previsível
- **Falha de projeto** (*design fault*)
 - erro no estado do projeto
 - imprevisível
- **Falha de ambiente**
 - mudanças indesejáveis e previsíveis do ambiente que afetam o sistema

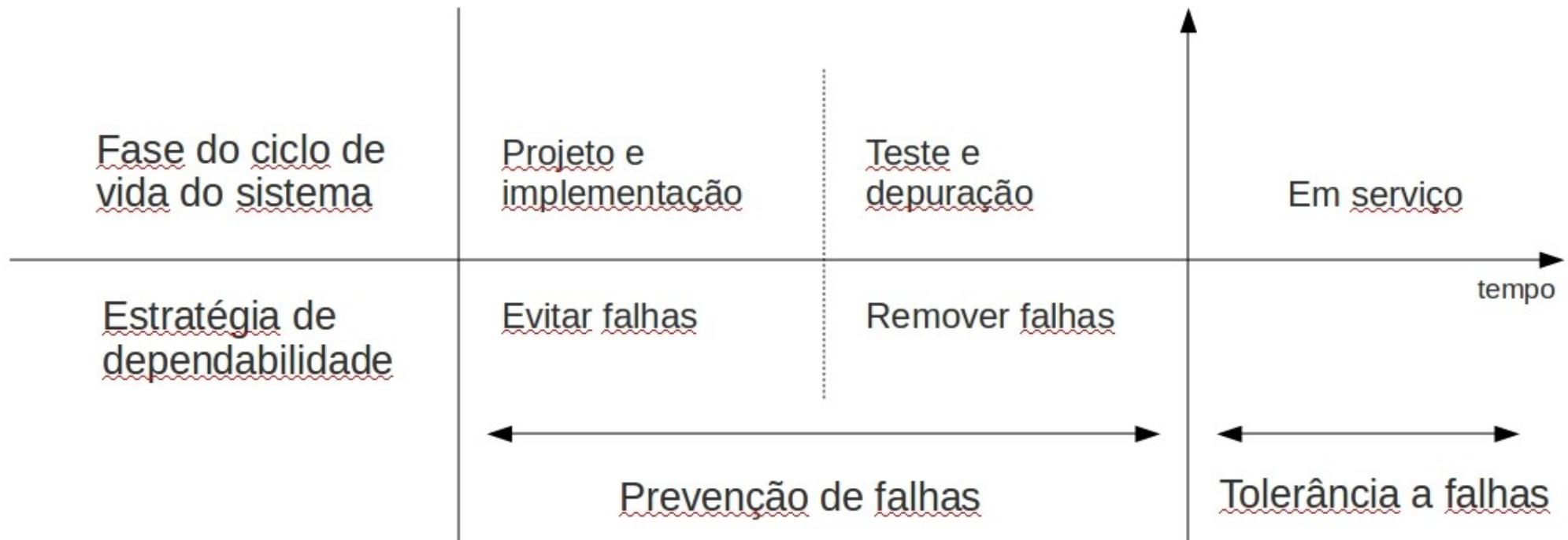


Estratégias de *dependability* (I)

- Dois enfoques complementares:
 - **Prevenção de falhas**: medidas preventivas que evitem a ocorrência de falhas.
Exemplos: testes, métodos formais, treinamento de pessoal, CMM, etc.
 - **Tolerância a falhas**: é a capacidade de um sistema preservar suas funcionalidades, ainda que parcialmente, mesmo na presença de falhas.



Estratégias de *dependability* (II)





Princípios da Prevenção de Falhas

- Tão importante quanto tolerar falhas é prevê-las.
- **Evitar falhas** - metodologias para evitar que falhas sejam introduzidas no desenvolvimento do sistema, e.g. métodos formais, programação estruturada.
- **Remover falhas** - validar os artefatos produzidos no desenvolvimento do sistema e remover suas falhas, e.g. testes.



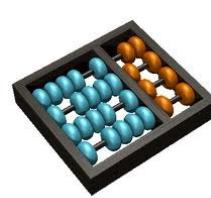
Verificação e Validação

- Verificação: "*Você está corretamente desenvolvendo o sistema?*"
- Validação: "*Você está desenvolvendo o sistema correto?*"



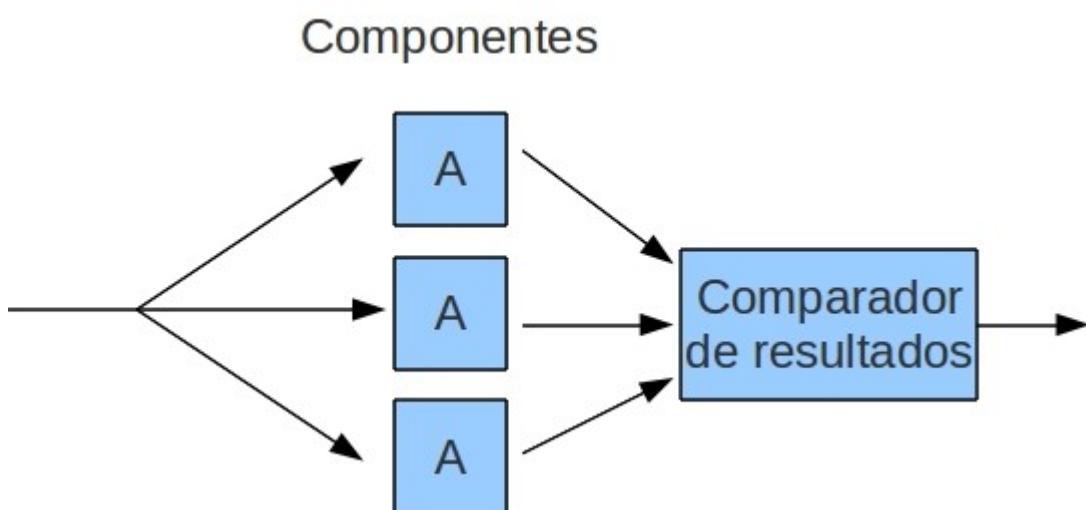
Software livre de falhas

- Hoje é possível desenvolver sistemas livre de falhas (*fault-free*), pelo menos sistemas pequenos [Sommerville, 2007, 8th edition].
- *Fault-free* significa que o software está de acordo com sua especificação.
- **Não** significa que ele irá sempre executar corretamente, já que sua especificação pode ter erros.
- O custo de produzir software livre de falhas é muito alto.
- + confiável => + tempo e + esforço para encontrar cada vez mais um número menor de falhas.
- É mais barato aceitar que o software poderá conter falhas residuais.



Técnicas de Tolerância a Falhas (TF)

- São baseadas no uso efetivo de **redundância**.
- Redundância (replicação) de dados e/ou processos
- Redundância temporal: *checkpoint/restart*
- Redundância de hardware: TMR - redundância modular tripla.

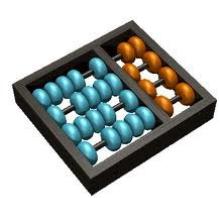




Redundância

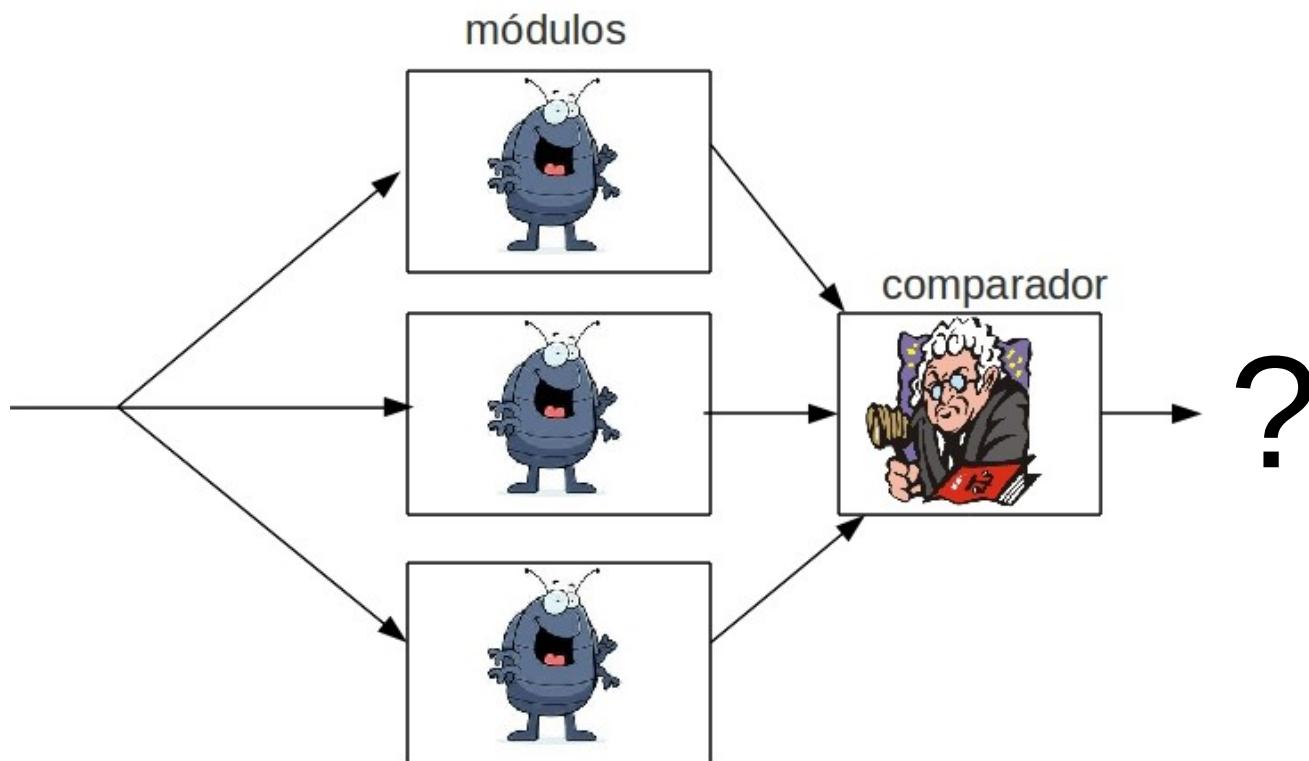
- Redundância são todos os elementos extras do sistema que não seriam necessários se fosse garantido que ele é livre de falhas.
- Redundância aumenta a complexidade e custo do sistema.

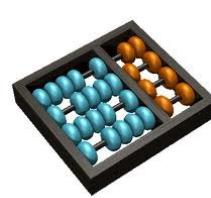




Diversidade de Projeto

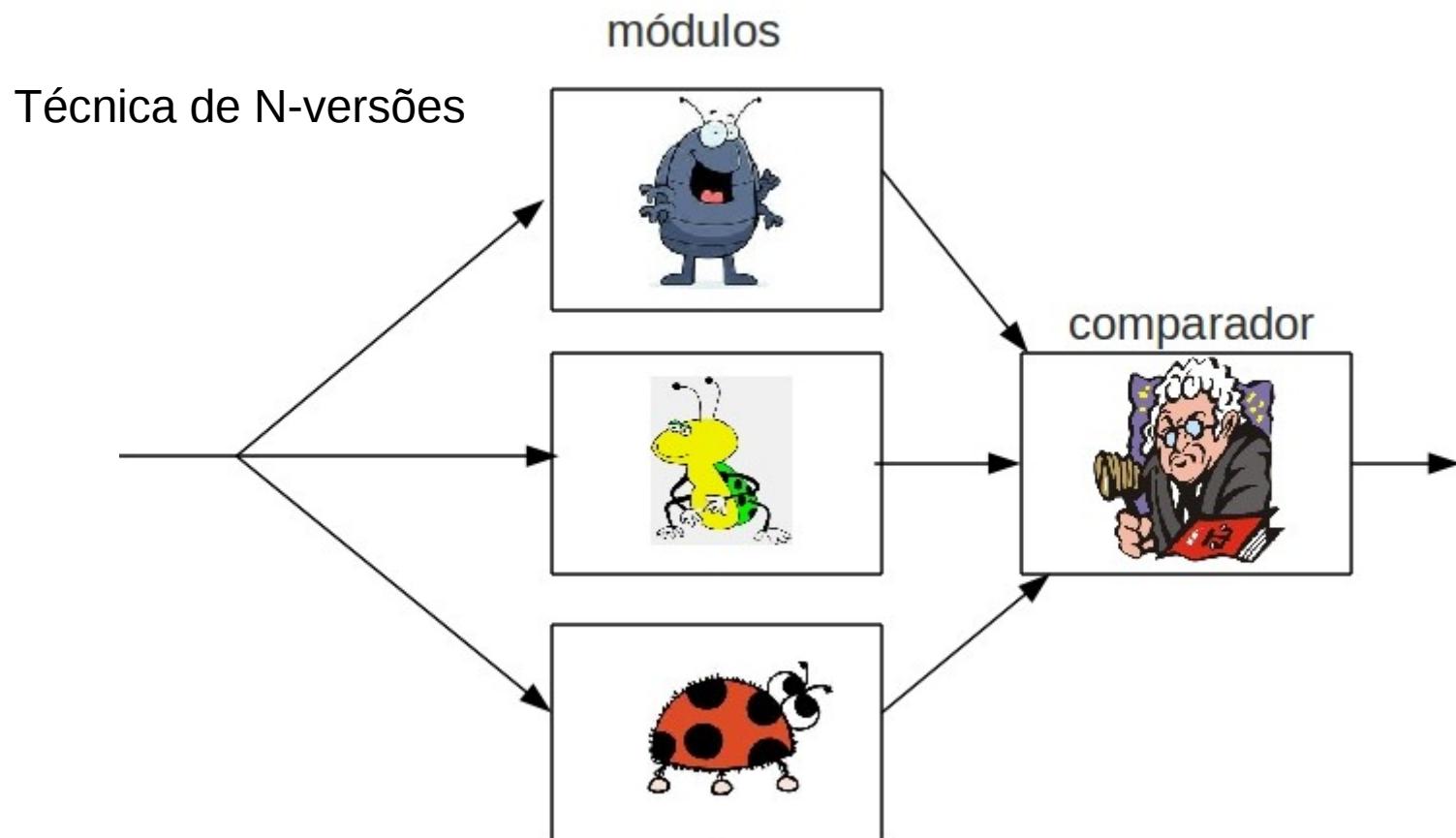
A replicação (cópia) de componentes de software multiplica *bugs* (falhas de projeto) no sistema.





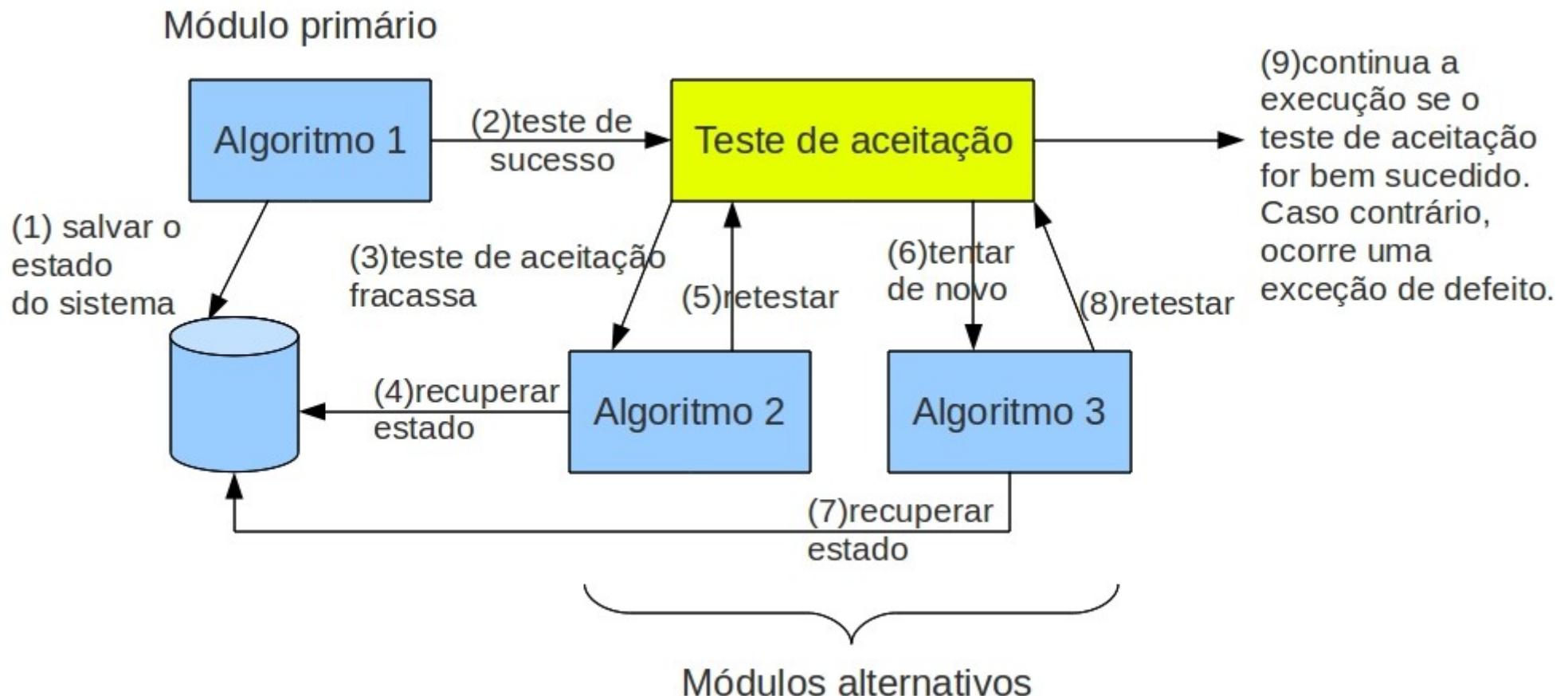
Diversidade de Projeto (II)

- Definição: uma mesma especificação de componente com várias implementações diversificadas (equipes diferentes, linguagens diferentes, algoritmos diferentes, etc).
- Apostila: implementações diferentes com *bugs* diferentes





Blocos de Recuperação





4 Fases de Tolerância a Falhas

- 1) Detecção de erros - o sistema deve detectar que um erro ocorreu.
- 2) Confinamento de danos - o sistema deve impedir que erros se propaguem.
- 3) Recuperação de erros - o sistema deve restaurar o seu estado para um estado livre de erros.
- 4) Tratamento da falha e serviço continuado - o sistema deve garantir que a falha não se repita => diagnóstico da falha e reparo/reconfiguração do sistema.



Recuperação de Erros por Retrocesso

- Recupera um estado anterior livre de erros.
- Ex.: transações atômicas em bancos de dados.
- Vantagens:
 - independe de avaliação de danos;
 - permite recuperar falhas arbitrárias;
 - pode ser aplicado a todos os sistemas.
- Desvantagem:
 - não simula o retrocesso do tempo



Recuperação de Erros por Avanço

- Manipula uma parte do estado atual para produzir um novo estado que seja livre de erros
- Técnica: uso de tratamento de exceções
- Vantagens:
 - custo de implementação mais baixo
 - melhor desempenho
- Desvantagens:
 - depende da avaliação de danos;
 - considera somente falhas antecipadas;
 - projetada para um sistema específico.
- Ex.: sistemas para foguetes, automação bancária, linhas de montagem, etc



A Explosão do Ariane 5

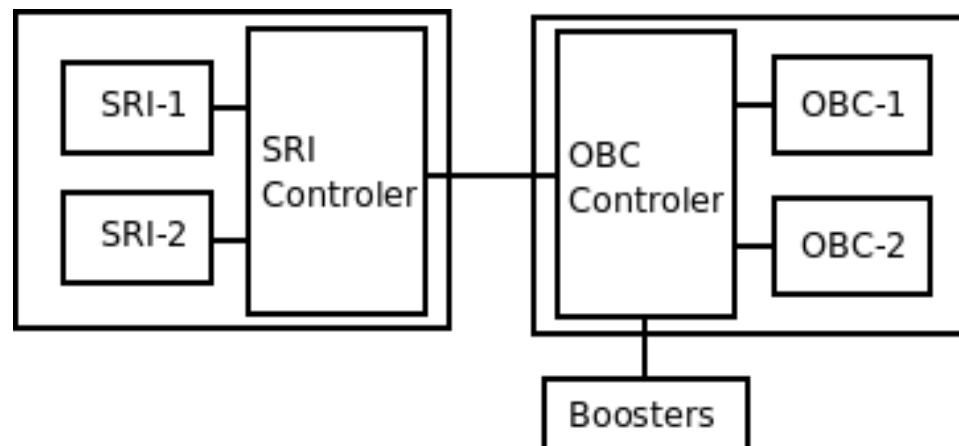
- Ariane 5 é um foguete lançador descartável usado para colocar satélites em órbitas geoestacionárias.
- Construído pela Agência Espacial Européia (ESA)
- Lançado em Junho de 1996 na base de Kourou na Guiana Francesa.
- Sucessor do bem sucedido Ariane 4.
- ~37s depois do lançamento, ele explodiu no ar em seu voo inaugural.
- Custo de aproximadamente **500 milhões de dólares**.

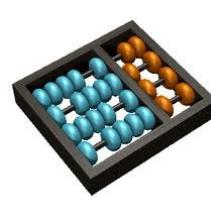




Ariane 5: Descrição do Sistema (I)

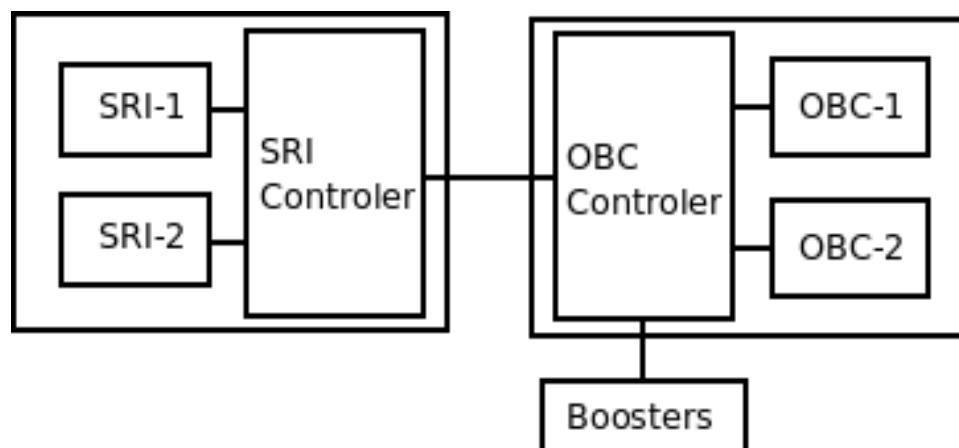
- A unidade SRI (Sistema de Referência Inercial) calculava o alinhamento do foguete.
- SRI foi reutilizada do foguete Ariane 4.
- O Ariane 5 tinha duas SRIs: uma ativa e outra em espera.
- Ambas SRIs eram **cópias idênticas**: mesmo hardware e mesmo software.





Ariane 5: Descrição do Sistema (II)

- O OBC (Computador de Bordo) também era duplicado, mas não era reutilizado.
- OBC usava os dados enviados pela SRI para controlar a trajetória do foguete através dos *boosters* (jatos propulsores).





Ariane 5: Cadeia de erros (I)

- 1) SRI-1 falha ao tentar armazenar um valor *float* de 64-bits em uma variável do tipo inteiro com sinal de 16-bits.
Motivo: a trajetória de voo do Ariane 5 tinha velocidade horizontal consideravelmente maior que a do Ariane 4
- 2) A SRI-2 é acionado, mas falha pelo mesmo motivo da SRI-1.
- 3) Não existe um SRI-3 para substituir a SRI-2.
- 4) Na falta de um outro módulo que funcionasse corretamente, a SRI-2 transmite uma sequência de bits de diagnóstico (*diagnostic bit patterns*) para OBC.



Ariane 5: Cadeia de erros (II)

- 4) OBC erroneamente interpreta os dados do SRI-2 como dados válidos.
- 5) OBC comanda que os bocais de deflexão sejam inclinados ao máximo.
- 6) O grau de inclinação dos bocais de deflexão criou uma carga aerodinâmica que separou os *boosters* do estágio principal.
- 7) O sistema de auto-destruição é corretamente acionado.
- 8) O foguete explode.



Video do Ariane 5

- Endereço no Youtube

[<http://www.youtube.com/watch?v=kYUrqdUyEpl>]



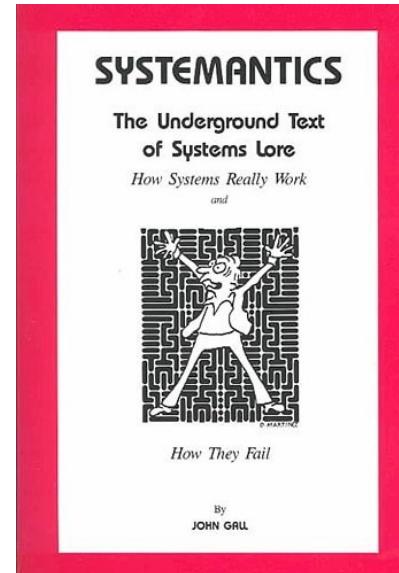
Ariane 5: Conclusões

- SRI foi reaproveitado do Ariane 4, cuja trajetória de lançamento e velocidade eram bem diferentes do Ariane 5.
- Adicionou-se proteção contra o erro causado pelo *overflow* em 4 das 7 variáveis da unidade SRI.
- **3 variáveis ficaram desprotegidas** para não sobrecarregar o processador do SRI a 80%.
- O cálculo do alinhamento feito pela unidade SRI após o lançamento era necessário para o Ariane 4 e **não** para o Ariane 5.
- Como esse cálculo após o lançamento não estava nos requisitos do Ariane 5, essa parte do software **não** foi testada.



Lições aprendidas

- "Sistemas complexos exibem comportamentos inesperados."
- "A realidade é muito mais complexa do que parece".
- "Um sistema complexo que funciona é invariavelmente oriundo da evolução de um sistema simples que funcionava".
- "Um sistema complexo se comporta como se tivesse vontade própria."



John Gall



Teorema Fundamental do Defeito



- "Um sistema pode falhar em um número infinito de formas".
- "Seu programa terá *bugs* e eles o surpreenderão quando você encontrá-los".
- "As variáveis cruciais são descobertas por acidente".
- "Quando um sistema grande falha, em geral o defeito exibido também é grande".
- "Quando um sistema *fail-safe* falha, ele falha por falhar em ser *fail-safe*".



Os defeitos podem ser úteis?

- Resposta: **Sim!**
- "*Cherish your exceptions, bugs, failures*".
- "Sucesso é uma questão de evitar as formas mais prováveis do sistema falhar".
- "Fracasso (*failure*) é talvez o nosso maior tabu"
- "Charles Darwin notou que nós seres humanos temos uma tendência para esquecer os fatos inconvenientes e, portanto, ele anotava suas observações imediatamente"



Parte 3: Tratamento de Exceções



Definição de Exceção

Aurélio: Exceção é um desvio da regra geral. Aquilo que se exclui da regra.

Tolerância a Falhas: Exceção é uma condição de erro detectada no sistema.

Ex.: se ao efetuar o saque de um caixa eletrônico, o sistema detecta que o cliente tem menos do que deseja sacar e lança (ou levanta) uma exceção de saldo insuficiente.

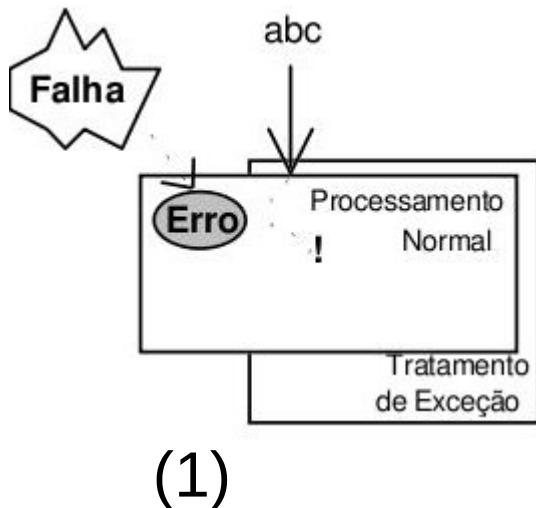


Funções do tratamento de exceções

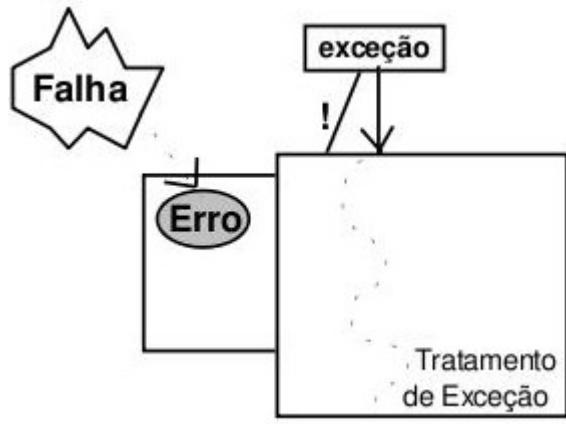
- (1) identificar a causa do erro;
- (2) determinar qual o agente da falha;
- (3) isolar o agente que falhou;
- (4) determinar a extensão do erro;
- (5) recuperar o estado do sistema;
- (6) reconfigurar o sistema;
- (7) reiniciar o processamento normal.



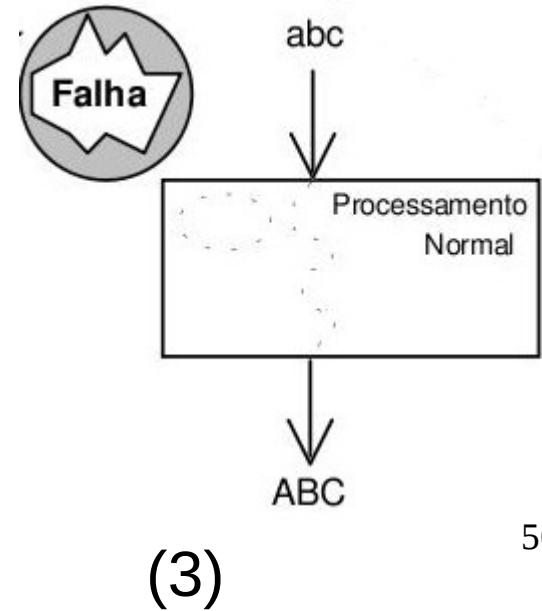
Exemplo



(1)



(2)



(3)

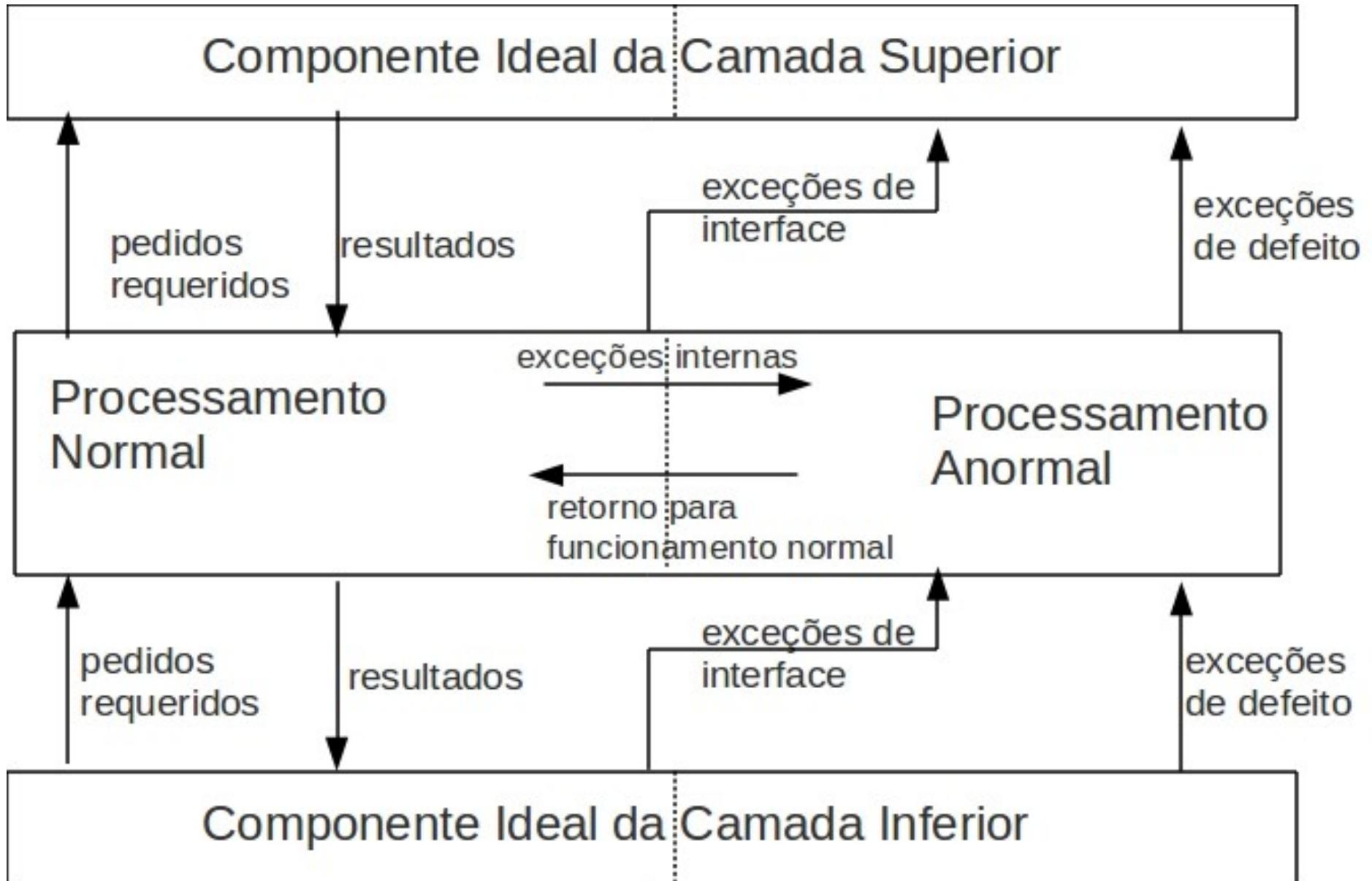


Tratamento de Exceções

- Adicionar o comportamento excepcional aumenta a complexidade do sistema (redundância implícita).
- O aumento da complexidade pode diminuir (ao invés de aumentar) a confiabilidade do sistema.
- O comportamento excepcional deve ser introduzido de forma estruturada.
- A atividade normal deve ser separada da atividade anormal (tolerância a falhas usando tratamento de exceções).



Componente ideal TF (I)





Componente ideal TF (II)

- Tratamento de exceções constitui um arcabouço genérico para implementar tolerância a falhas (escola de Newcastle).
- Camadas e recursão são estruturas apropriadas para incorporar TF.
- Uma camada trata falhas provenientes da camada imediatamente inferior.



Componente ideal TF (III)

- O modelo de falhas do sistema pode ser dividido em modelos menores, mais fáceis de serem tratados.
- **Modelo de falhas**: é a representação do conhecimento das possíveis falhas e como elas influenciam no processo.



Tipos de exceções

- **Exceções de interface** são geradas por uma solicitação ilegal (e.g. solicitar um serviço inexistente).
- **Exceções de defeito** devem ser lançadas se um componente se este não conseguir prover um serviço especificado.
- **Exceções internas** são geradas pelo componente para invocar seu tratadores internos.

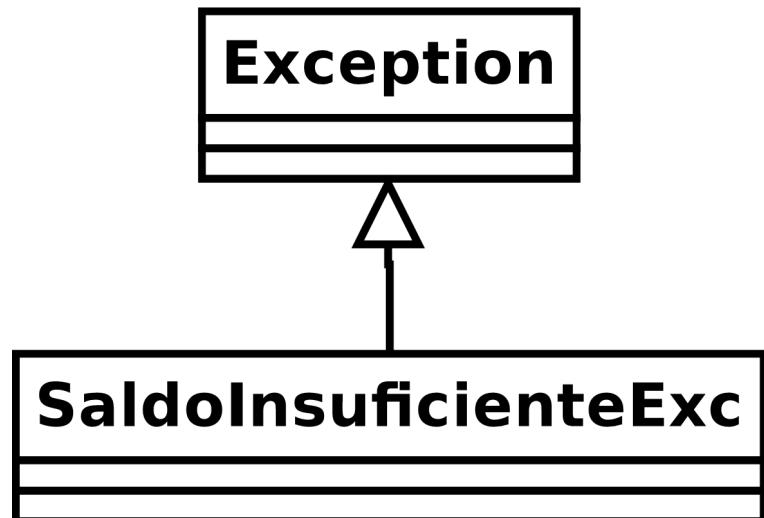


Exceções em Java

- Exceções em Java são objetos da classe `Exception`.
- Esses objetos podem ser lançados.

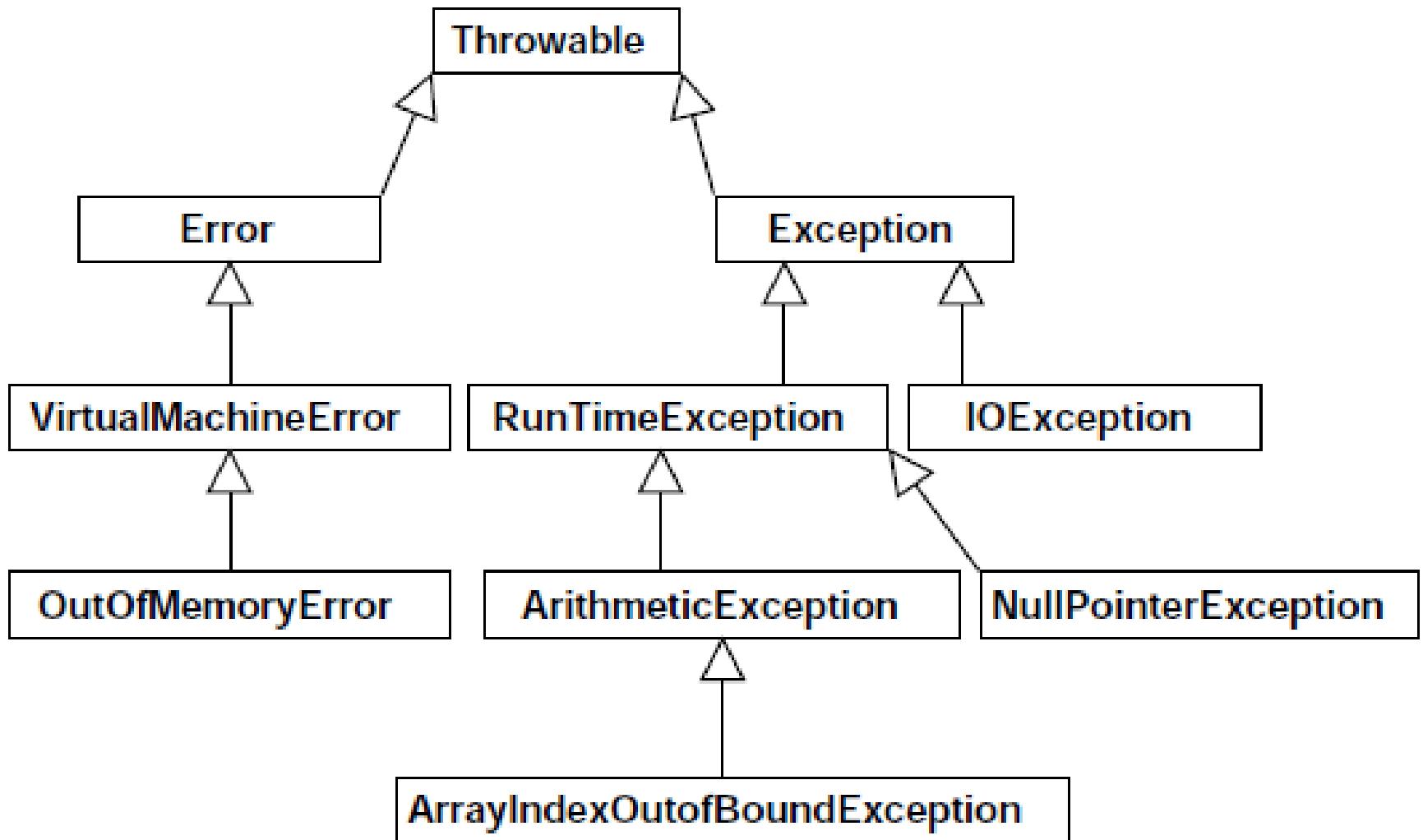
Exemplo

```
if (condição de erro X){  
    throw new ContaCorException("erro detectado");  
}
```





Hierarquia de Exceções





Tratadores em Java

- Quando uma exceção é lançada o método que estava sendo executado é interrompido.
- O processamento é desviado para um tratador (*handler*)

```
//código...

try {

    // código que pode lançar MyException

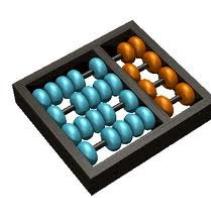
} catch (SaldoInsuficienteExc e1) {

    //tratador (handler) da exceção

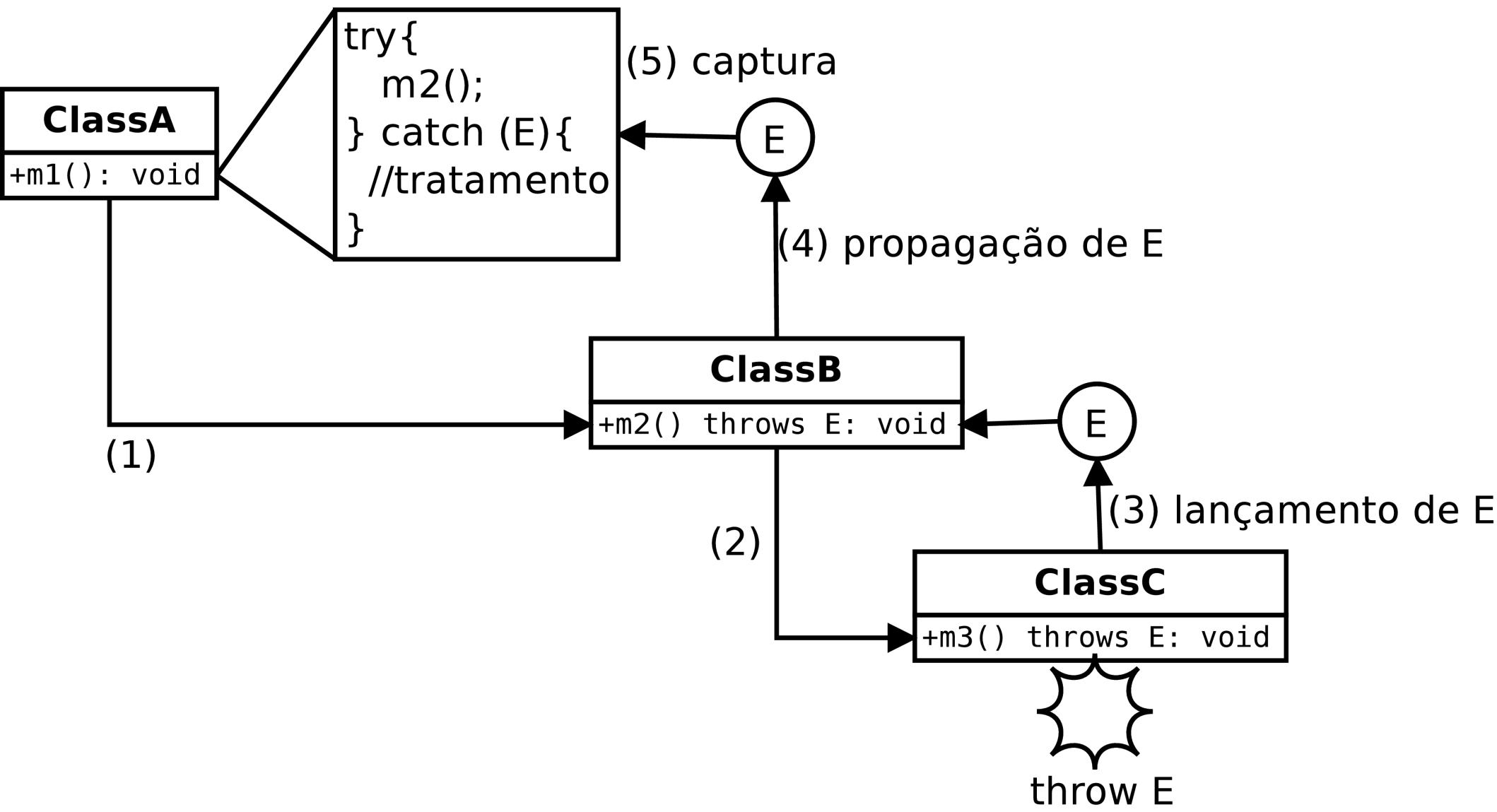
}

//mais código...


}
```



Propagação de exceções





Exceções: Verificadas x Não-verificadas

- **Exceções verificadas** são obrigadas a serem tratadas pelo programa e surgem em situações específicas e bem definidas.
- Objetos do tipo `Exception` e seus subtipos.
- **Exceções não-verificadas** não são obrigadas a serem tratadas pelo programa.
- Objetos do tipo `Error`, seus subtipos e `RuntimeException`



Continuação do Fluxo de Controle

- **Modelo de terminação** assume que quando uma exceção é lançada, o tratador correspondente lida com a exceção e completa o bloco de execução.
- **Modelo de *resumption*** que o tratador recupera o estado do programa e continua a execução a partir da operação que lançou a exceção.



Um teste simples

- Quantas vezes a bola foi passada entre as pessoas que usavam camiseta branca?





Estiveram atentos?

- Minha resposta: Não sei!
- Você não viu o macaco?
- Vamos ver de novo...





Exceções em Java

- Java contribuiu muito para que o comportamento excepcional não fosse um "gorila invisível".
- Será que os desenvolvedores usam os mecanismos de tratamento de exceções de Java corretamente?



Desenvolvedores estão tratando exceções adequadamente? (I)

- Não é uma tarefa de alta prioridade

I ignore exceptions because I don't need to take care of them. Whenever something goes wrong just fix the error, it's not worth spending the time.



I care only about the right path.





Desenvolvedores estão tratando exceções adequadamente? (II)

- Uso forçado de tratamento de exceções

I don't love to use it [exception handling] but for some features of the language, I have to use it.

I need to catch exceptions because Java requires me to do it. I just fulfill the language's requirements.





Desenvolvedores estão tratando exceções adequadamente?

- Exceções ignoradas (*swallowed*): não são re-lançadas e nem tratadas.

// código...

```
try {
    // código que pode lançar MyException
} catch (MyException e1) {}
// mais código...
}
```

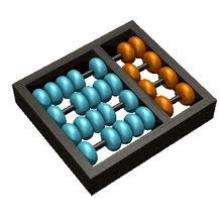
- Razão: parar de escrever "*throws*" nas assinaturas das operações.

[D. Reimer and H. Srinivasan. Analyzing exception usage in large java applications - ECOOP03 workshop on EHS]



Exceções são úteis?

- Alguns pesquisadores não estão convencidos...
- Andrew P. Black. *Exception Handling: the case against*. Universidade de Oxford, 1982.



Alternativas para exceções

When to use exceptions:

- [UseExceptionsInsteadOfErrorValues](#)
- [DontUseExceptionsForFlowControl](#)
- [AvoidExceptionsWheneverPossible](#)
- [CodeWithoutExceptions](#)

Alternatives to Using Exceptions

- [NullObjects](#)
- [ErrorValues](#)
- [PassAnErrorHandler](#)
- [InvisibleExceptionHandlers](#) (more or less an [AntiPattern](#))
- [SoftwareTransactionalMemory](#)
- [UseAssertions](#)
 - [WhatAreAssertions](#)
 - [DoNotUseAssertions](#)
 - [AssertionsAsComments](#)
 - [CountInAssertions](#)
- [ExceptionalValue](#)
- [BottomPropagation](#)
- [ReplaceEmptyCatchWithTest](#)
- [LookBeforeYouLeap](#)
 - But [CoupleLeapingWithLooking](#)



Desafios

- Convencer desenvolvedores que soluções *ad-hoc*s são menos poderosas que mecanismos de tratamento de exceções.
- Criar processos sistemáticos para tratar exceções durante o desenvolvimento.
- Criar técnicas para modelar exceções e fluxos excepcionais em nível arquitetural.
- Definir padrões para divulgar as boas práticas para construir o comportamento excepcional.
- Criar novos sistemas de tratamento de exceções para aplicações contemporâneas.



Parte 4: Arquitetura de Software



Arquitetura de Software

- Define a estrutura de alto nível sistema, mostrando sua organização geral como um conjunto de componentes e conectores.
- Sobe a montanha e vê a floresta
- Ênfase nas interações entre componentes.
- Condiciona atributos de qualidade (propriedades sistêmicas). Ex.: *dependability*, adaptabilidade, desempenho, etc.



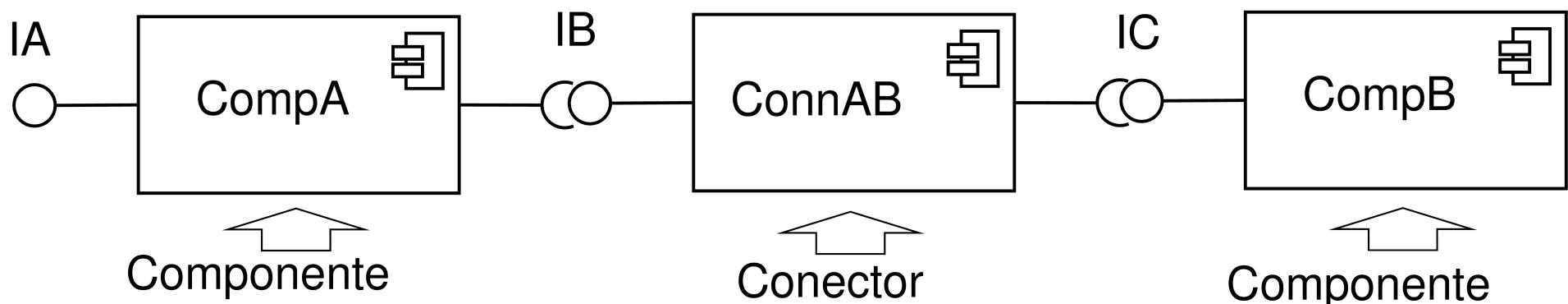
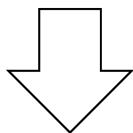
Componente, conector e configuração

- **Componentes Arquiteturais** (módulos ou componentes) que executam a computação (requisitos funcionais do sistema)
- **Conectores Arquiteturais** explícitos que unem os componentes e coordenam as interações entre os componentes para que a composição satisfaça restrições e atributos de qualidade globais do sistema específico a ser construído.
- **Configuração Arquitetural** representa um conjunto de componentes e conectores interligados.



Exemplo de Arquitetura

Configuração arquitetural





Funções da Arquitetura de Software



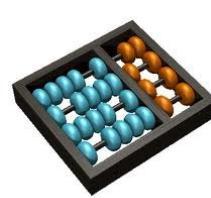
- Registrar decisões de projeto
- Mapear requisitos em componentes
- Permitir análise de riscos
- Preservar a integridade do sistema durante a evolução
- Criar um artefato reutilizável



Funções da Arquitetura (II)

A motivação principal para a definição de uma arquitetura de software é:

- Proporcionar um plano concreto com o objetivo de predizer o comportamento do sistema antes dele ser construído e guiar o seu desenvolvimento, sua manutenção e sua evolução através do ciclo de vida do sistema.
- A importância do papel da arquitetura começou a ser reconhecido por desenvolvedores e pesquisadores em meados da década de 90

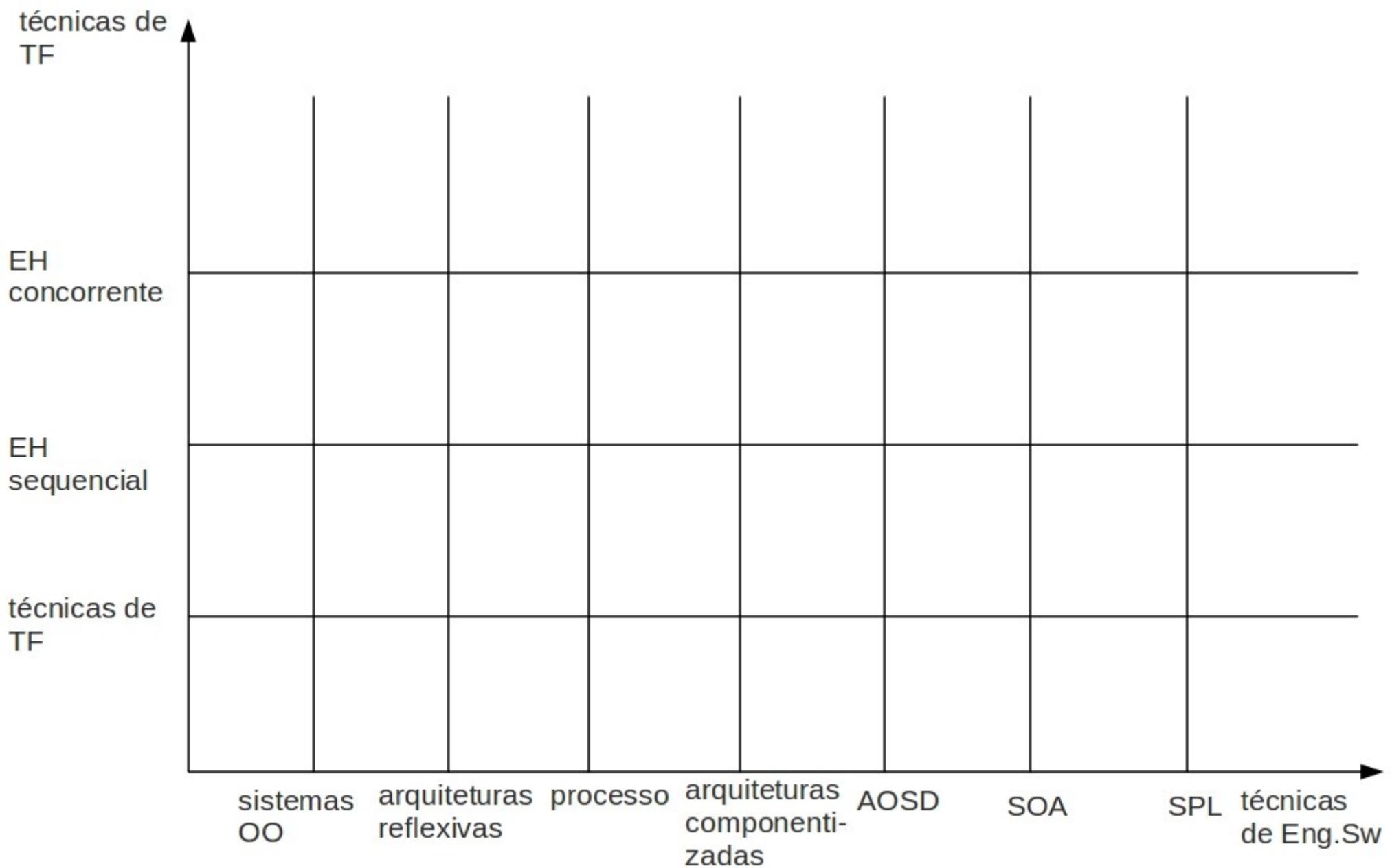


Funções da Arquitetura (III)

- A arquitetura serve como uma ferramenta importante para comunicação, para tomada de decisões (*reasoning*), para análise e para o crescimento de sistemas;
- A arquitetura de software é o resultado de um conjunto de decisões de negócio e técnicas;
- Portanto, ela é um resultado de influências (explícitas e implícitas): técnicas, de negócios e sociais;



TF & Engenharia de Software





Parte 5: Desenvolvimento centrado na arquitetura usando tratamento de exceções

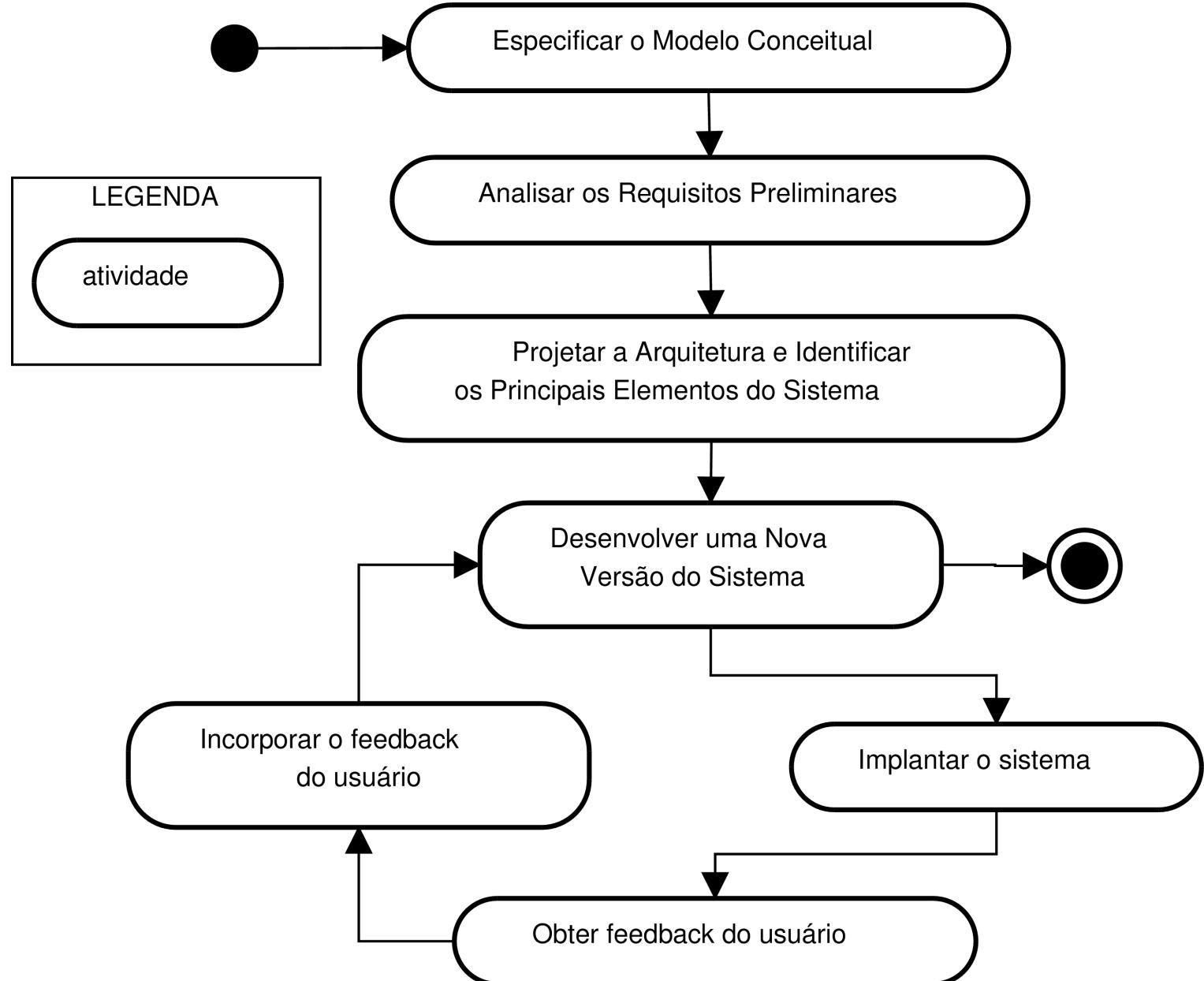


Etapas do Desenvolvimento centrado na Arquitetura

- 1) Criação de um plano de negócios;
- 2) Compreensão dos requisitos do sistema;
- 3) Criação ou seleção da arquitetura do sistema;
- 4) Representação e divulgação da arquitetura;
- 5) Avaliação da arquitetura;
- 6) Implementação do sistema baseado na arquitetura;
- 7) Evolução do sistema.



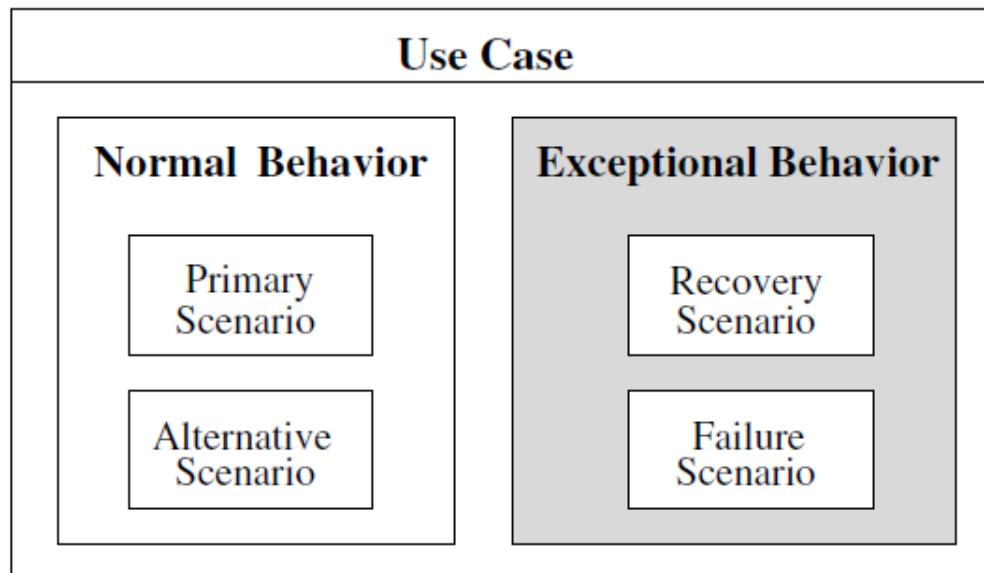
Processo centrado na Arquitetura





Exceções no Desenvolvimento de Software (I)

- Abordagem sistemática para incorporar o tratamento de exceções ao longo das fases de desenvolvimento.
- O comportamento excepcional é descrito por cenários excepcionais, que podem ser de defeito ou recuperação.





Caso de Uso DrenarReservatório

Cenário Recuperável 1

Violação de invariante

- Sinal:

- Tratador:

- Pós-condições:

Durante a extração de água o sensor FluxoAgua acusou ausência de fluxo, o que significa mal funcionamento da bomba; sensor FluxoAgua desligado e Bomba ligada iniciar o caso de uso BombaFalhou
Bomba ligada, sensor FluxoAgua ligado

Cenário Recuperável 2

Violação de pós-condição

- Sinal:

- Tratador:

- Pós-condições:

Após ter desligado a bomba, o sensor de fluxo FluxoAgua permanecia ligado, o que indica que a bomba não foi corretamente desligada;
sensor FluxoAgua ligado e Bomba desligada
Tentar desligar a bomba mais uma vez; Enviar alarme para operador.
Bomba desligada, sensores FluxoAgua desligado e PoucaAgua ligado.

Cenário de Falha 1

Violação de invariante

- Sinal:

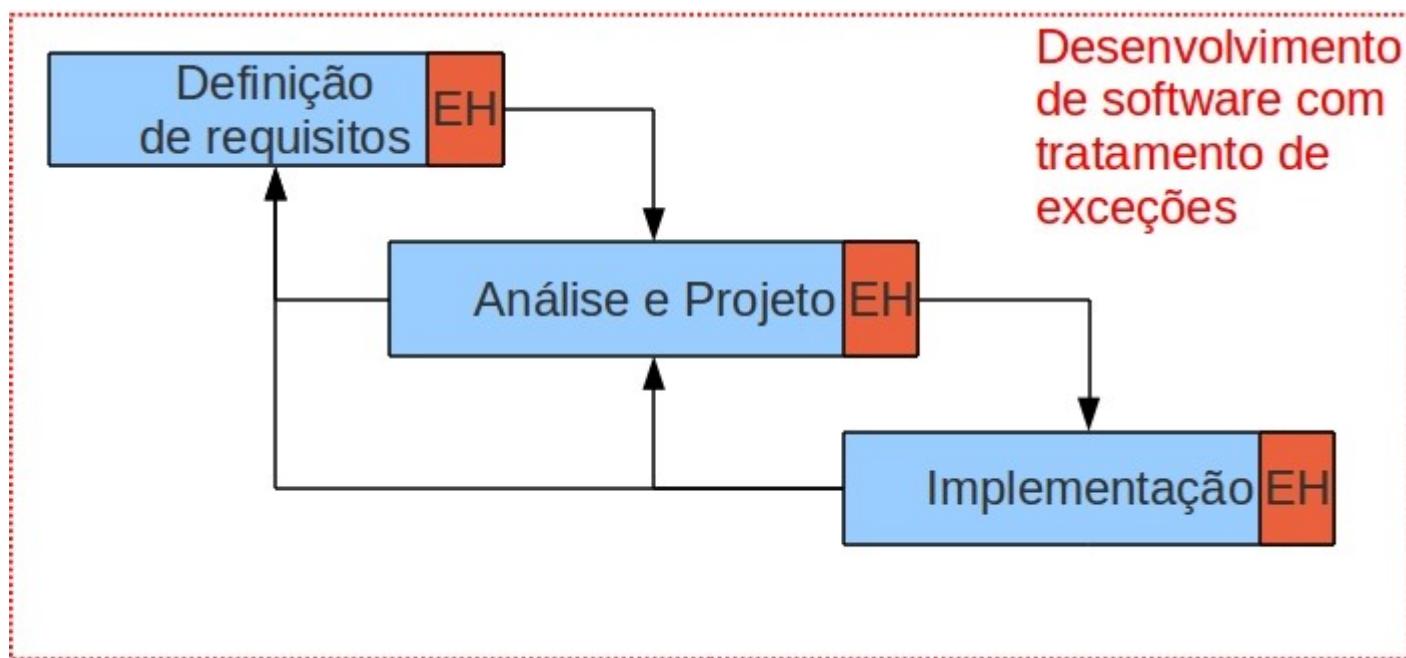
- Tratador:

- Pós-condições:

Durante a extração de água o sensor de metano acusou alto nível do gás metano no ambiente, o que propicia risco de explosão da mina;
MuitoMetano ligado e Bomba ligada
Desligar bomba; Enviar alarme para operador.
Bomba desligada e MuitoMetano ligado



Desenvolvimento com Tratamento de Exceções





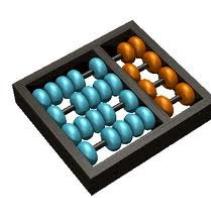
Etapa 3 - Criação da Arquitetura do Sistema (I)

- Uso de Padrões Arquiteturais;
- Criatividade.
- Integridade Conceitual e Consistência;
- Simplicidade.



Etapa 3 - Criação da Arquitetura do Sistema (II)

- Frederick P. Brooks Jr., livro “The Mythical Man-Month: Essays on Software Engineering – Anniversary Edition”, Addison-Wesley, 1995, 1975, pai do IBM 360.
- Brooks argumenta que a “integridade conceitual” é a chave para uma arquitetura sólida de sucesso;
- Integridade conceitual reflete a harmonia do todo, isto é, reflete um (e somente um) conjunto de ideias de projeto que funcionam harmoniosamente, ao invés de conter muitas ideias boas mas que são independentes e desacordadas e no conjunto não se casam.



Definição de Padrão Arquitetural

- “Um padrão arquitetural descreve um esquema básico de organização para estruturar sistemas de software, definindo um conjunto de componentes, suas responsabilidades, e estabelecendo regras e diretrizes para organizar o relacionamento entre esses componentes.” [Buschman, 1996]



Elementos de um Padrão

Um padrão (arquitetural, de análise, de projeto, etc.) é descrito através das seguintes informações:

- 1) Nome;
- 2) Problema;
- 3) Solução;
- 4) Consequências;
- 5) Exemplos (opcional).



Padrões Arquiteturais

- Arquiteturas OO (*Call-Return*);
- Camadas (*Layers*);
- Reflexão (*Reflection*);
- MVC (*Model-View-Controller*);
- *Pipes and Filters*;
- Quadro Negro (*Blackboard*);
- *Broker* (ou *broadcast*);



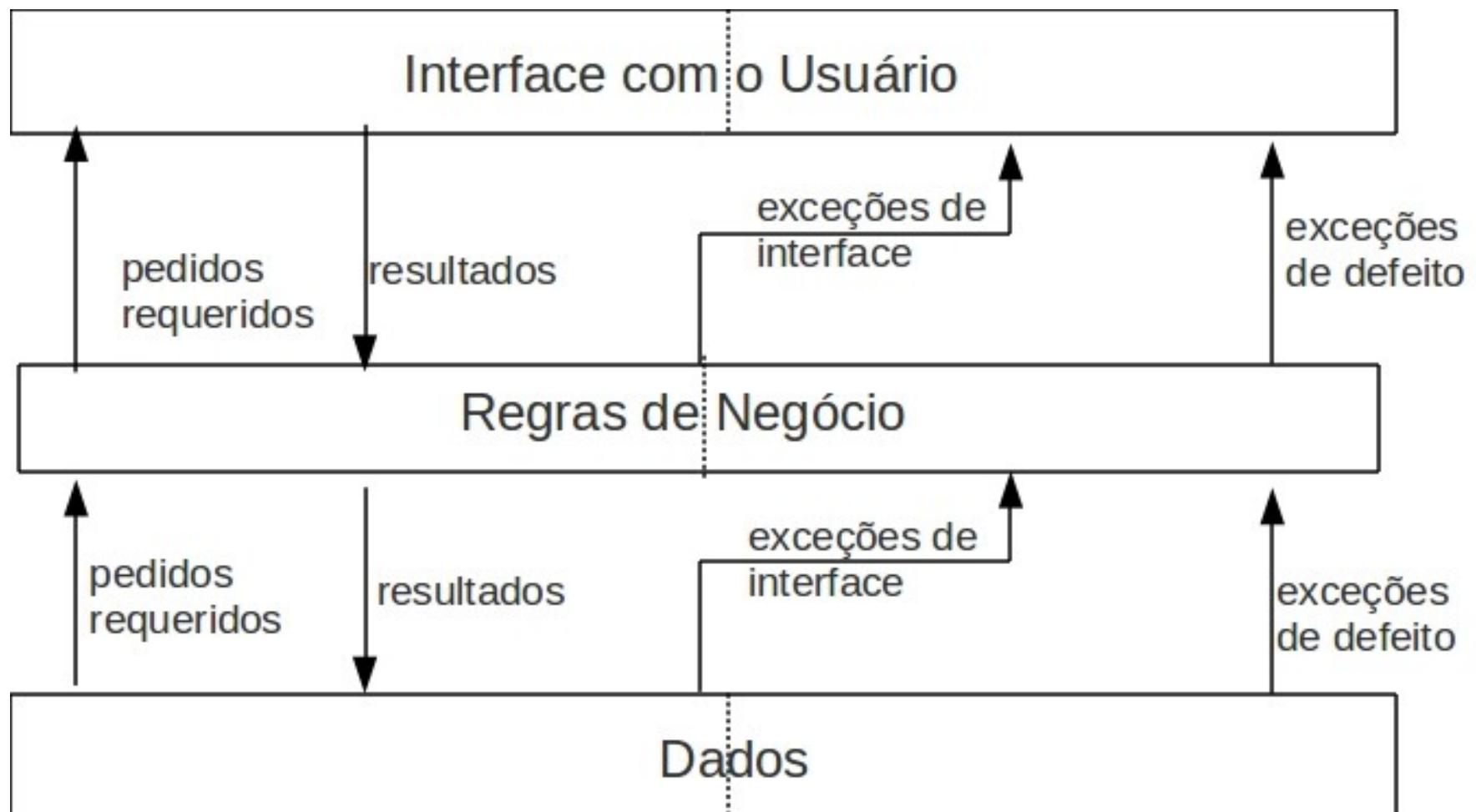
Padrão Arquitetural Camadas

- É um tipo de *call-return*;
- Componentes:
 - São camadas com um alto nível de abstração.
- Conectores:
 - Fluxos de comunicação entre camadas.
- Restrições
 - Uma camada somente recebe requisição de uma camada imediatamente superior;
 - Uma camada somente envia requisição para uma camada imediatamente inferior.



Padrão Arquitetural

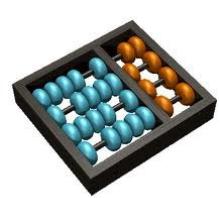
Camadas [usar componente ideal]





Arquiteturas OO e Exceções

- Início dos anos 90: linguagens de programação como Smalltalk-83, C++, Ada, Eiffel, etc.
- Estudo comparativo de mecanismos de tratamento de exceções existentes em linguagens de programação OO.
- Classificação das decisões de projeto para construir mecanismos efetivos para serem usados na prática
- Java contribuiu muito para que o comportamento excepcional não passasse desapercebido.



Garcia, Rubira,
Romanovsky, and Xu.
A comparative study of
exception handling
mechanisms for
building dependable
object-oriented
software. Journal of
Systems and Software
2001.

Taxonomy Aspects		Design Decisions		Exception Mechanisms		Ada 95		Lore		Smalltalk		Eiffel		Modula3		C++		Java		Delphi		Guide		Ext. Ada		BETA		Arche						
A1.	Exception Representation	Symbols		-1		-1		-1		-1		-1		-1		-1		-1		-1		-1		-1		-1		+1						
		Data Objects															+1		+1		+1													
A2.	External Exceptions in Signatures	Full Objects		+1																											+1			
		Unsupported		-1		-1		-1											-1											-1				
A3.	Separation between Internal and External Exceptions	Optional		0													0														0			
		Compulsory															+1															+1		
A4.	Attachment of (+) Handlers	Hybrid																	+1															
		Unsupported		-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1			
A5.	Handler Binding	Supported																																
		Statement		+1	+1												+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1		
A6.	Propagation of (+) Exceptions	Block		-1													-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1		
		Method															+1											+1	+1					
A7.	Continuation of (+) Control Flow	Object																																
		Class		+1	+1	+1	+1																											
A8.	Cleanup Actions	Exception		+1																														
		Unsupported																															-1	
A9.	Reliability Checks (+)	Use of Explicit Propagation		0					0			0																				0		
		Specific Construct			+1	+1				+1			+1				+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1			
A10.	Concurrent Exception Handling	Automatic Cleanup																																
		Dynamic Checks		+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1				
A10.	Concurrent Exception Handling	Static Checks															+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1			
		Unsupported		-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1					
A10.	Concurrent Exception Handling	Limited		0																														
		Complete																																
Final Score		1	5	3	1	3	3	3	6	3	3	6	3	3	6	3	7	-1	3	6	3	7	-1	3	6	3	6	3	6	3	6			

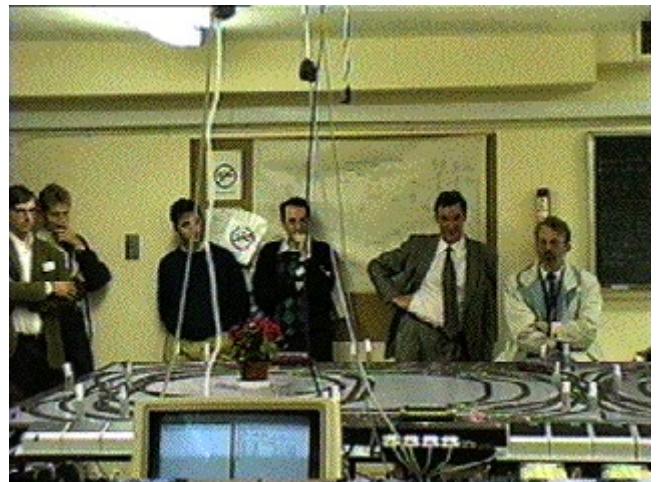


Arquiteturas OO e TrainSet

- Falhas de ambiente
- Controlador de software para um grande modelo de linhas de trem
- 92 *switches* e 150 sensores.

(1) Detecção e recuperação de erro;

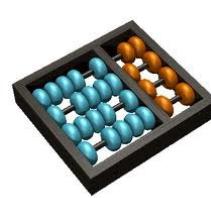
(2) Tratamento de falhas





Arquiteturas OO para Tratamento de Exceções

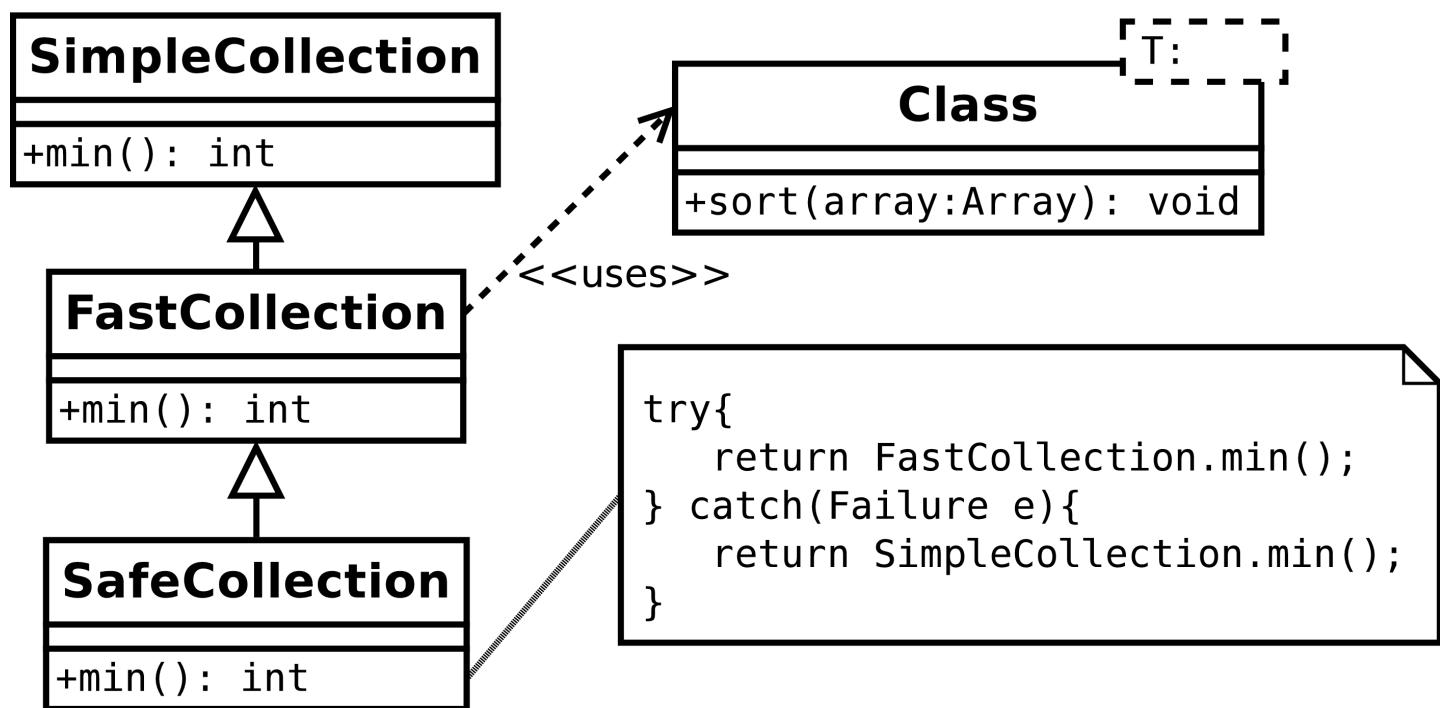
- Sistemas de tratamento de exceções:
 - reduzem o esforço de desenvolvimento.
 - apóia representação de erros como exceções.
 - definição de *handlers* em contextos adequados.
 - emprega a estratégia adequada de tratamento de exceções.



Recuperação de erros em Arquiteturas OO

- Uso de herança e tratamento de exceções para implementar em C++ a recuperação de erros por avanço e o componente ideal TF.
- Polimorfismo paramétrico para implementar blocos de recuperação usando recuperação de erros por retrocesso.

Rubira and Stroud. Forward and Backward error recovery in C++. Journal of Object-Oriented systems, 1994.





Parte 6: Tratamento de exceções concorrentes (cooperativas)



Reflexão na água





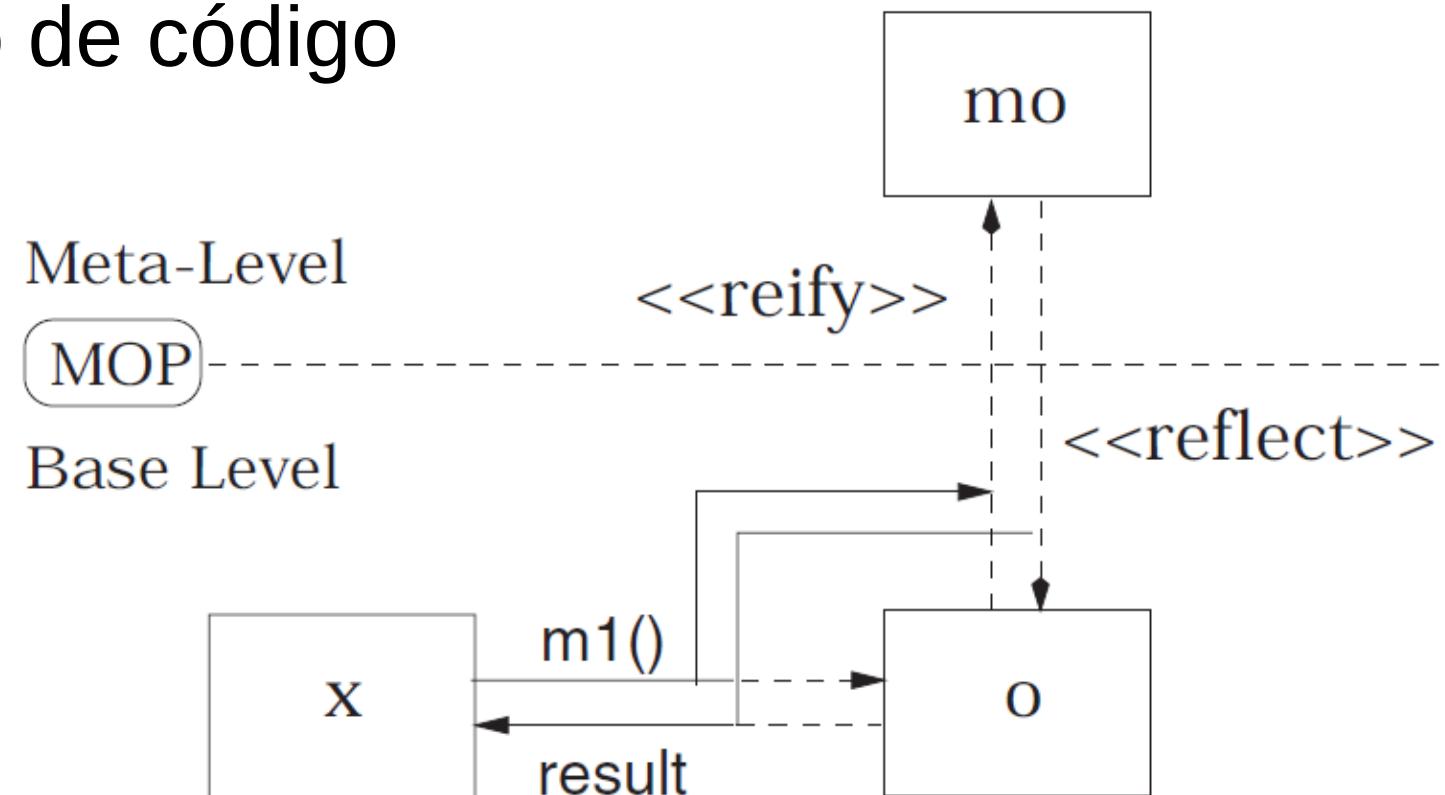
Reflexão Computacional (I)

- É capacidade de observar e manipular o comportamento computacional de um sistema através de um processo chamado de **reificação**.
- O sistema incorpora estruturas que representam ele próprio.
- Permite interceptar e modificar os efeitos das operações numa aplicação: e.g. criar um objeto, invocar um método.



Reflexão Computacional (II)

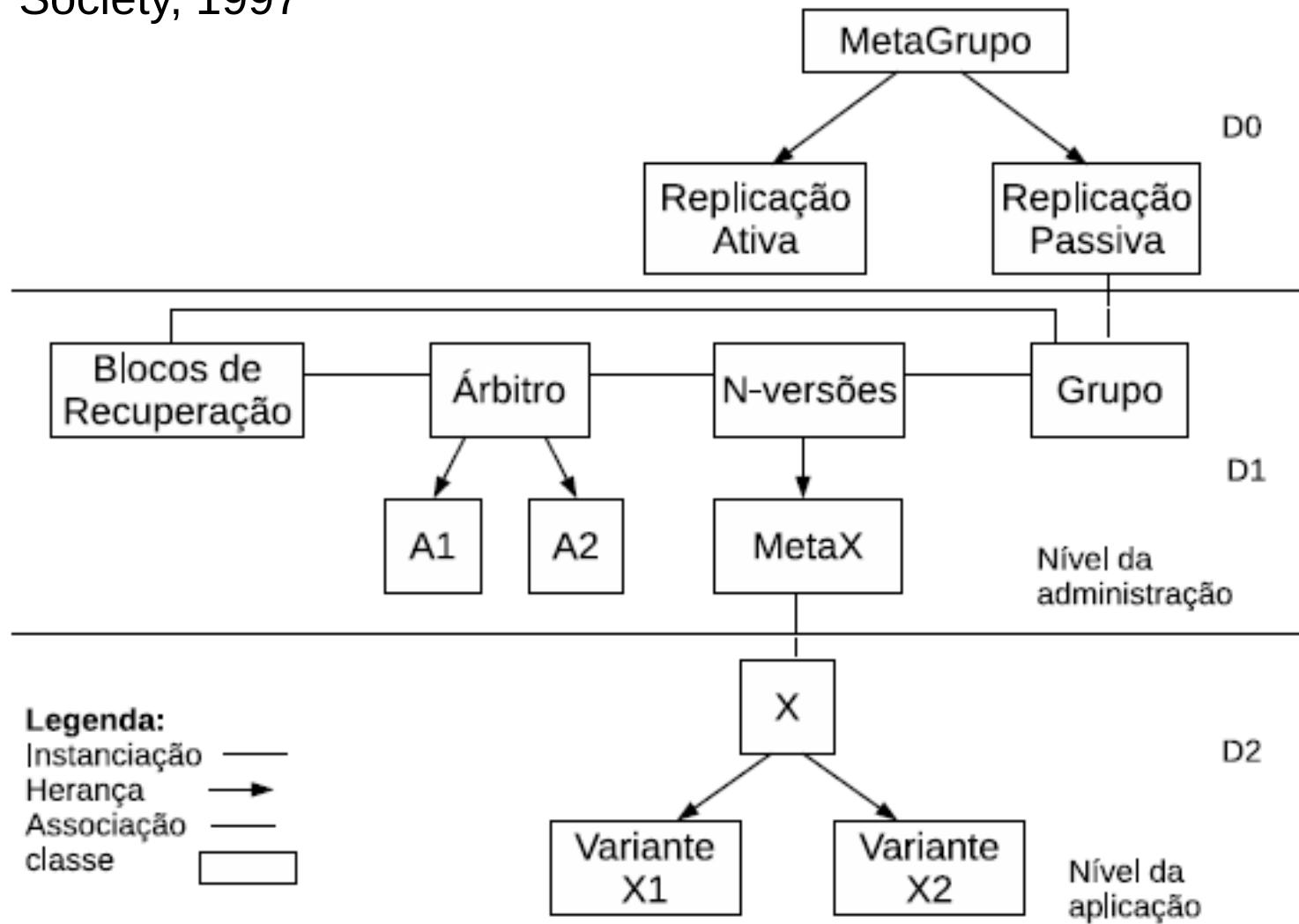
- Separação de interesses: objetos de aplicação e de gerenciamento.
- Transparência de forma não-intrusiva
- Reutilização de código





Arquiteturas reflexivas (I)

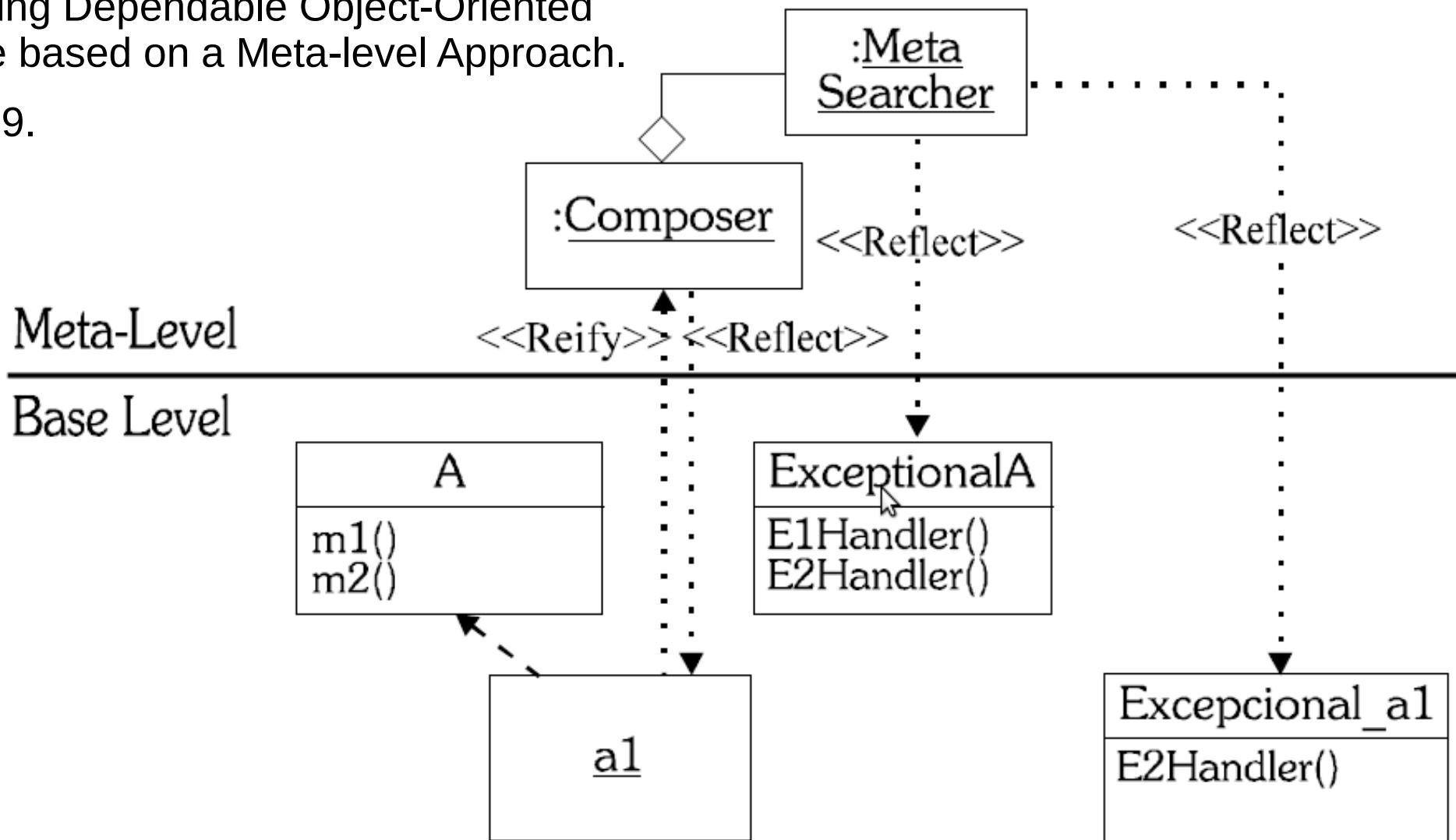
L.E.Buzato, C.M.F.Rubira & M.L.B.Lisbôa. A Reflective Object-Oriented Architecture for Developing Fault-Tolerant Software. Journal of the Brazilian Computer Society, 1997





Arquiteturas reflexivas (II)

A.Garcia, D.M.Beder & C.M.F.Rubira. An Exception Handling Mechanism for Developing Dependable Object-Oriented Software based on a Meta-level Approach.
ISRRE'99.





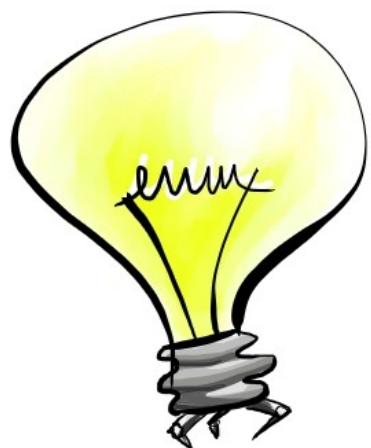
Arquiteturas de Software baseadas em Componentes

- Componentes reutilizáveis, que visam atender múltiplos sistemas
 - maior complexidade
 - contexto de utilização desconhecido
- Especificações incompletas e incerteza de comportamento
- Componentes "de prateleira" com código inacessível e qualidade duvidosa



Componente Arquitetural X Componente de Software

- Uma unidade de software fornecido independentemente e que oferece serviços através de interfaces.
 - Interfaces especificadas contratualmente
 - Dependências de contexto explícitas
 - Desenvolvido e utilizado independentemente



60W
127 V





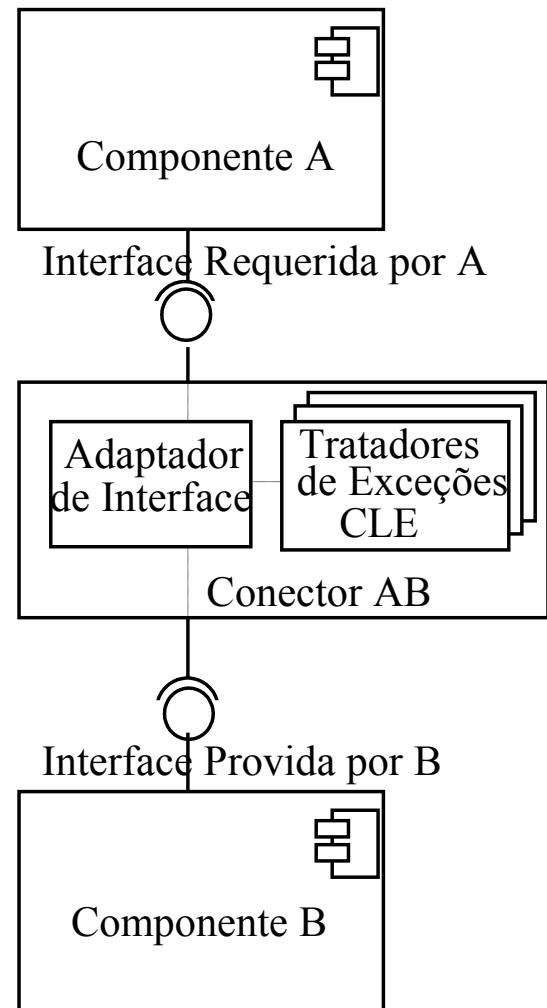
Tratamento de Exceções em Arquiteturas baseadas em Componentes

- Facilitar a reutilização dos novos componentes de software desenvolvidos
- Adaptar o comportamento excepcional de componentes já existentes ao contexto de um novo sistema tolerante a falhas



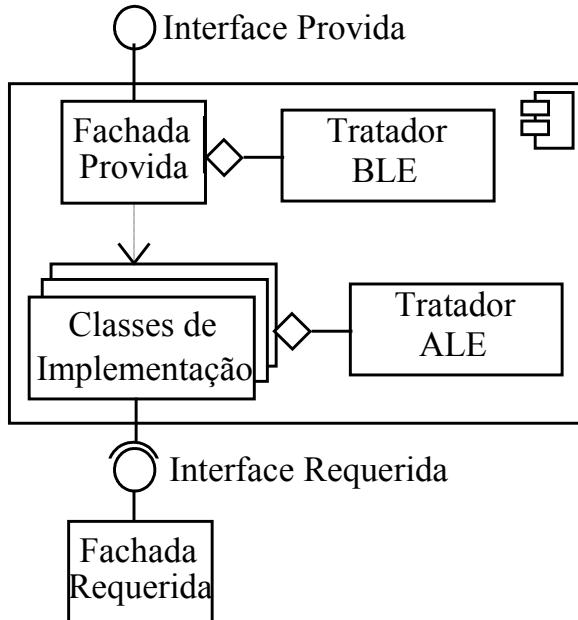
Estratégia Inter-Componente

- Conectores
 - Tratar exceções de configuração
 - Adaptar interfaces excepcionais
 - Resolver conflitos entre as hipóteses de falhas

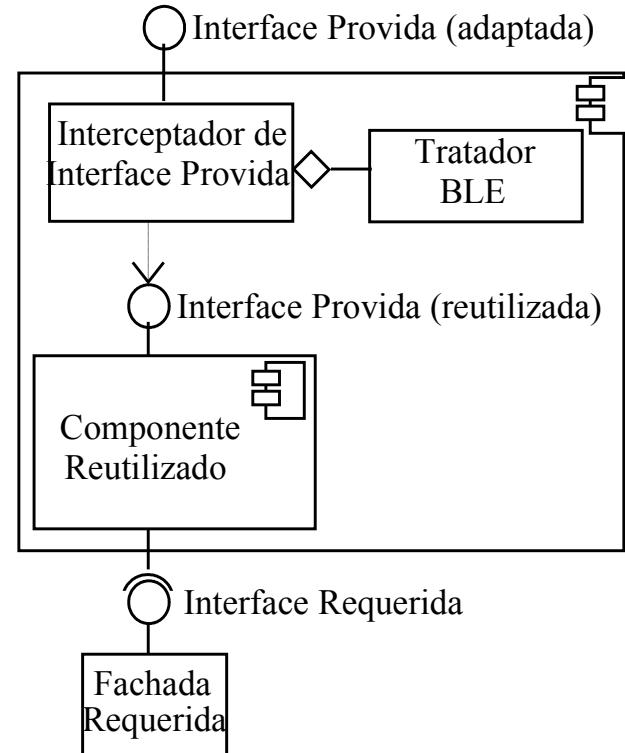




Estratégia Intra-Componente



Componente
desenvolvido do
“zero”

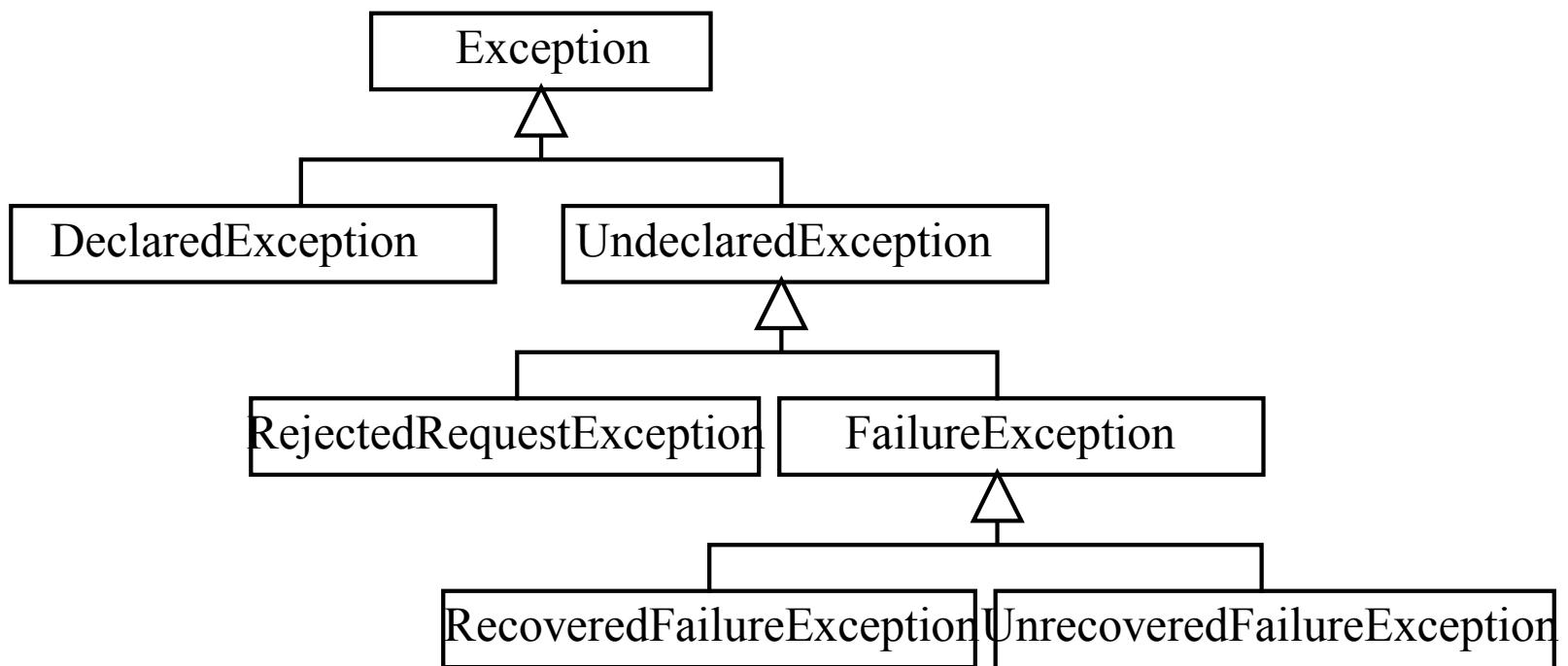


Componente já
existente

ALE = Application-Level Exception
BLE = Boundary-Level Exception



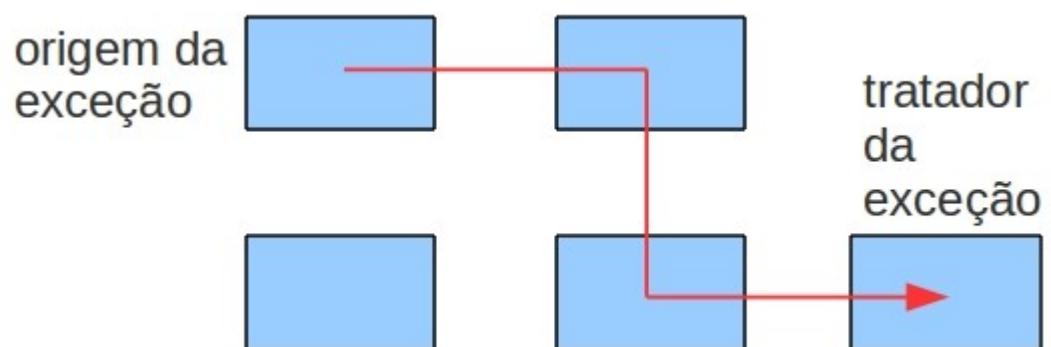
Hierarquia de exceções arquiteturais

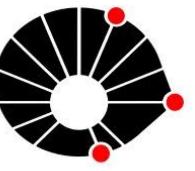




Fluxos excepcionais em arquiteturas (I)

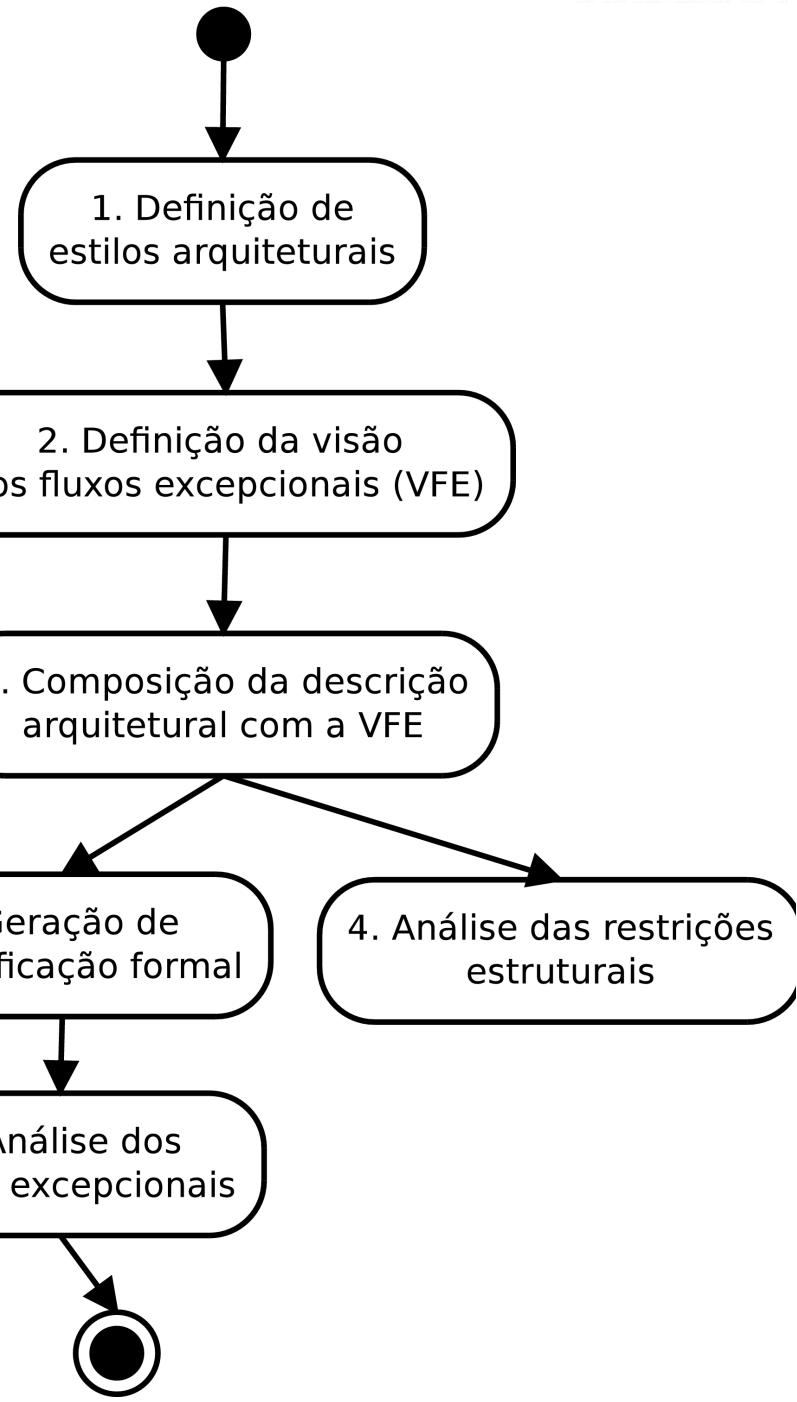
- Especificação de fluxos excepcionais:
 - Atribuição de responsabilidade (sinalização, mascaramento, recepção e propagação de exceções)
 - Sem ambiguidades
 - Ortogonal à descrição "normal" da arquitetura
 - Caixas e linhas
 - Padrões arquiteturais
 - Análise automatizada





Fluxos excepcionais em arquiteturas (II)

- Ao final do processo, o arquiteto saberá se o fluxo especificado é válido ou um contra-exemplo será mostrado.

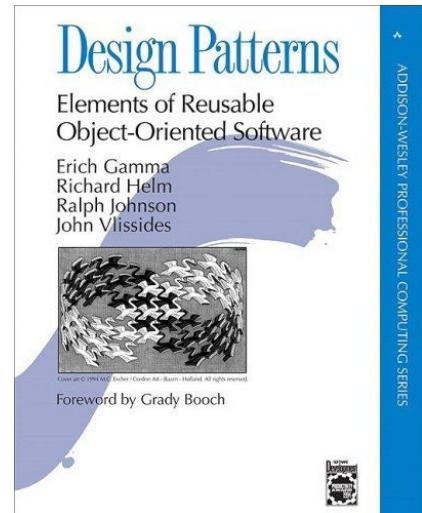




Padrões de Projeto para Tratamento de Exceções



- *"Para desmistificar o projeto de tratamento de exceções, deve-se oferecer técnicas, diretrizes e padrões."*
- Wirfs-Brocks. Designing for Recovery. IEEE Software, 2006.
- Wirfs-Brocks. Toward exception handling best practices and patterns. IEEE Software, 2006.

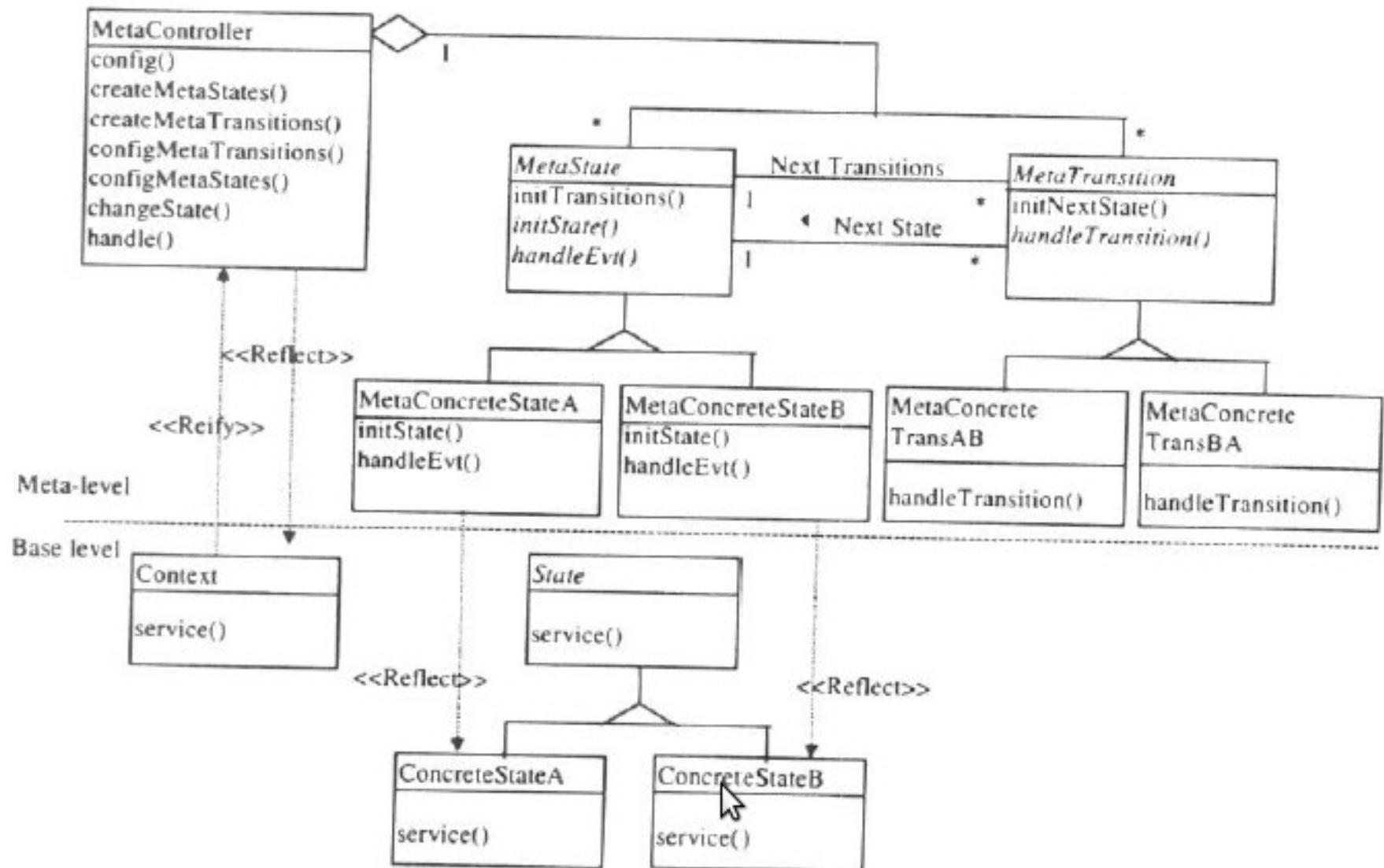


Exception
Handling
Design
Patterns?



Padrões de projeto para TF

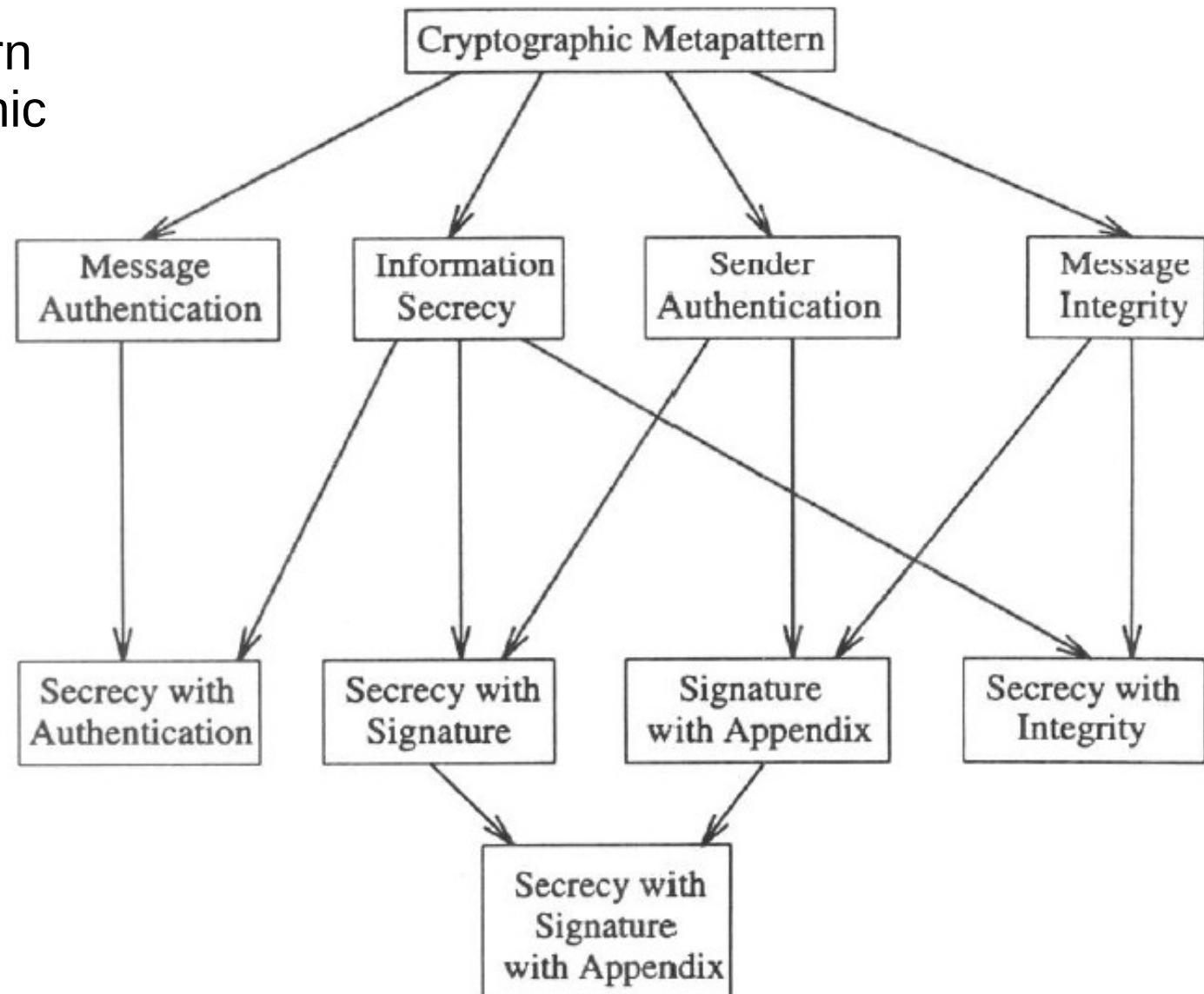
[L.L.Ferreira & C.M.F.Rubira. The Reflective State Pattern. PloP'98]





Padrões de Projeto para Security

[A.Braga, C.M.F.Rubira & R.Dahab. Tropyc: A Pattern Language for Cryptographic Software. PloP'98]





Sistemas OO Concorrentes

- Sistemas OO concorrentes consistem de várias *threads* (ou processos) executando métodos de forma concorrente em objetos.
- Exceções levantadas por um processo podem afetar outros processos, o que torna
- **Concorrência Competitiva** - dois ou mais objetos projetados separadamente não se conhecem e competem por recursos comuns (e.g. transações atômicas de banco de dados).
- **Concorrência Cooperativa** - dois ou mais objetos executam uma tarefa juntos e explicitamente trocam informações para atingir um objetivo comum (e.g. ações atômicas).



Transação Atômica

- É uma ação lógica que realiza uma sequência de operações básicas em objetos compartilhados
- Efeito tudo ou nada.
- São usadas para tolerar defeitos de hardware como *crashes* de nós ou defeitos de comunicação.
- Usadas para banco de dados

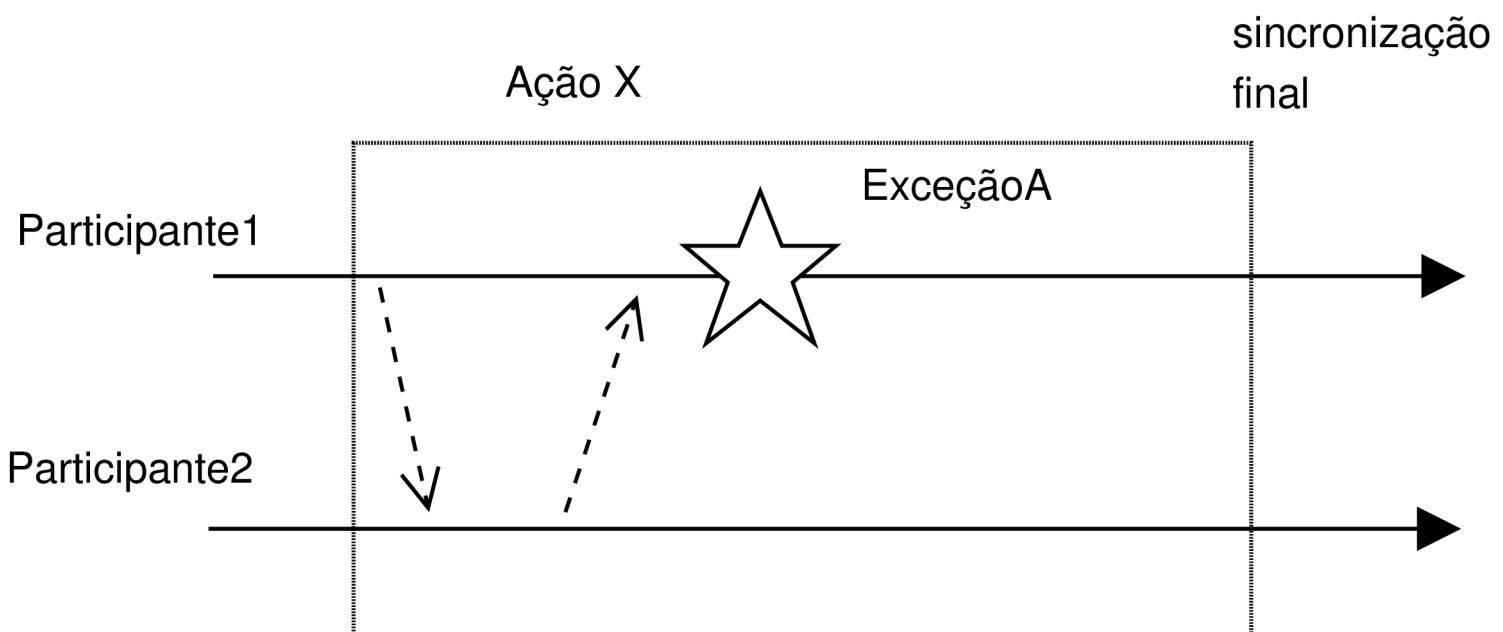


Ação Atômica

- É uma unidade de estruturação que modela o padrão de interações entre objetos (estrutura dinâmica).
- A atividade de um grupo de objetos constitui uma ação atômica se não existirem interações entre o grupo e o resto do sistema até que ela termine.
- A execução de sistemas concorrentes usa ações atômicas para definir regiões de recuperação de tal forma que erros não se espalhem.
- Se um erro é detectado dentro de uma ação, seus participantes executam uma recuperação coordenada.
- Hardware faults, software design faults, e environmental faults.



Ação Atômica





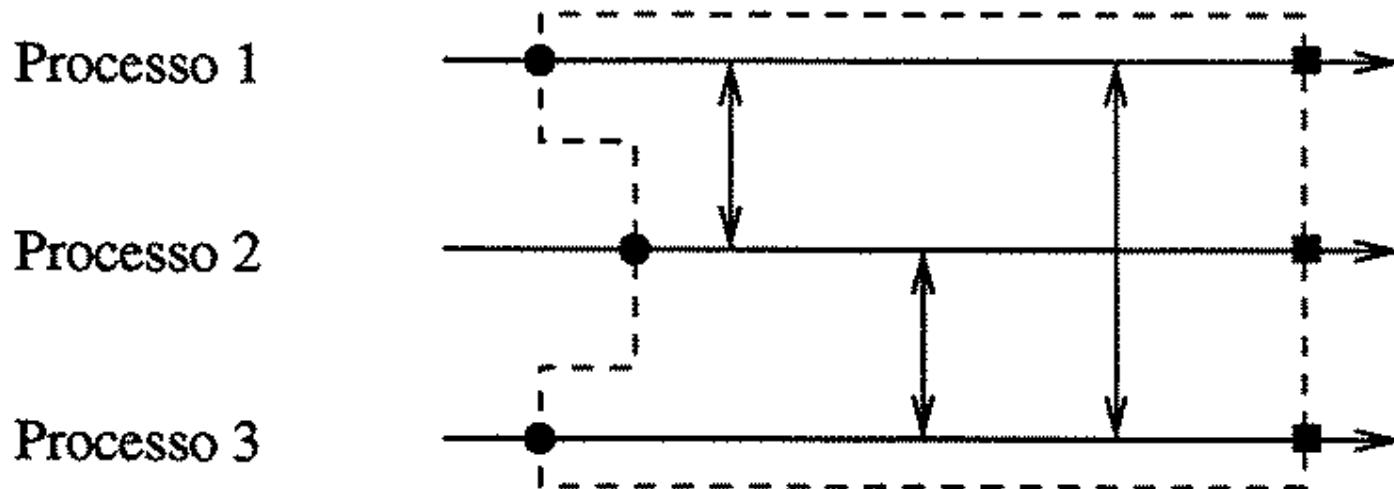
Conversação (I)

- É um esquema de ação atômica usado para diversidade de software.
- Seus participantes restauram seus estados para tolerar falhas de projeto (blocos de recuperação).
- Cada participante deve estabelecer seu ponto de recuperação.
- Se um participante falha em seu teste de aceitação, uma exceção é levantada.
- O estado original de cada participante é restaurado.
- Cada participante usa uma versão alternativa da implementação.



Conversação (II)

- Fronteira: linha de recuperação, linha de teste e duas linhas verticais de isolamento.
- Entrada pode ser assíncrona
- Saída síncrona => teste de aceitação



- Ponto de Recuperação
- Teste de Aceitacão



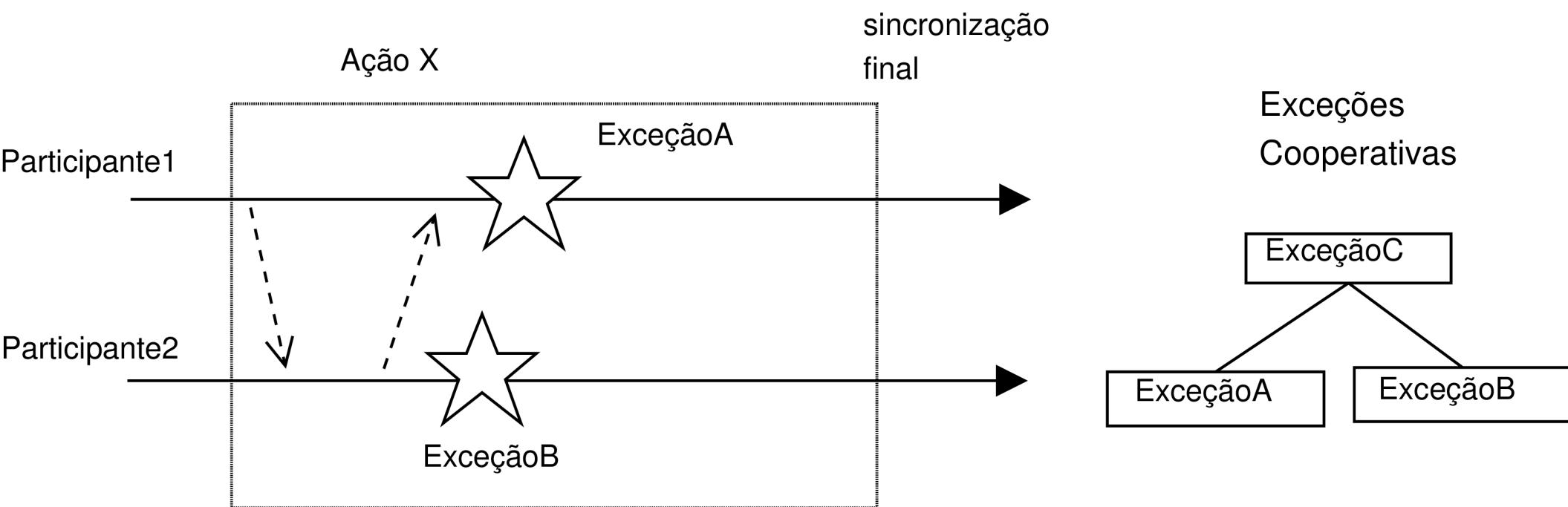
Recuperação Coordenada de Erros



- Se uma exceção é levantada em um dos participantes, tratadores para a mesma exceção serão iniciados em todos os participantes.
- Estes tratadores são projetados cooperativamente pelos desenvolvedores.
- Um esquema de resolução de exceções deve ser usado para combinar múltiplas exceções em uma única se elas foram lançadas ao mesmo tempo.
- Os tratadores podem usar uma mistura de técnicas de recuperação de erros por avanço e por retrocesso.



Árvore de Resolução





Ações Atômicas Coordenadas

- Integra ações atômicas e transações atômicas em um mesmo arcabouço conceitual.
- Ações atômicas são usadas para controlar concorrência cooperativa e implementar recuperação coordenada de erros .
- Transações atômicas são usadas para controlar concorrência competitiva e manter a consistência dos recursos compartilhados na presença de falhas.



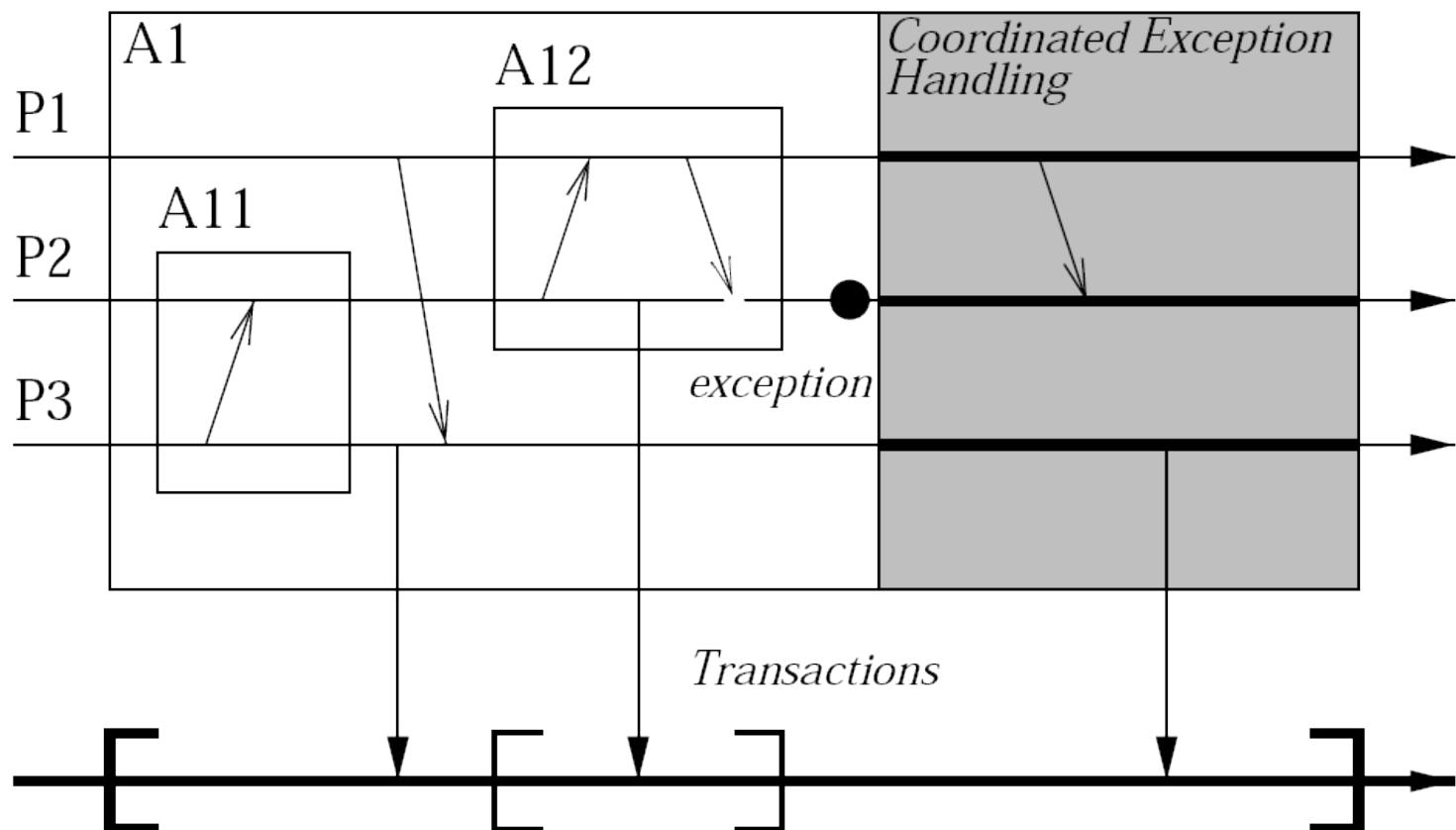
Ações Atômicas Coordenadas

- Consiste por um conjunto de papéis que são ativados concorrentemente pelos participantes.
- As ações começam quando todos os papéis forem ativados e termina quando todos alcançam o fim da ação.
- Os participantes da ação cooperam explicitamente através de objetos locais.
- Objetos externos são transacionais, ou seja, a sequência de operações realizadas por uma dada ação atômica coordenada em um conjunto de objetos externos deve ser atômica em relação às outras ações atômicas coordenadas.



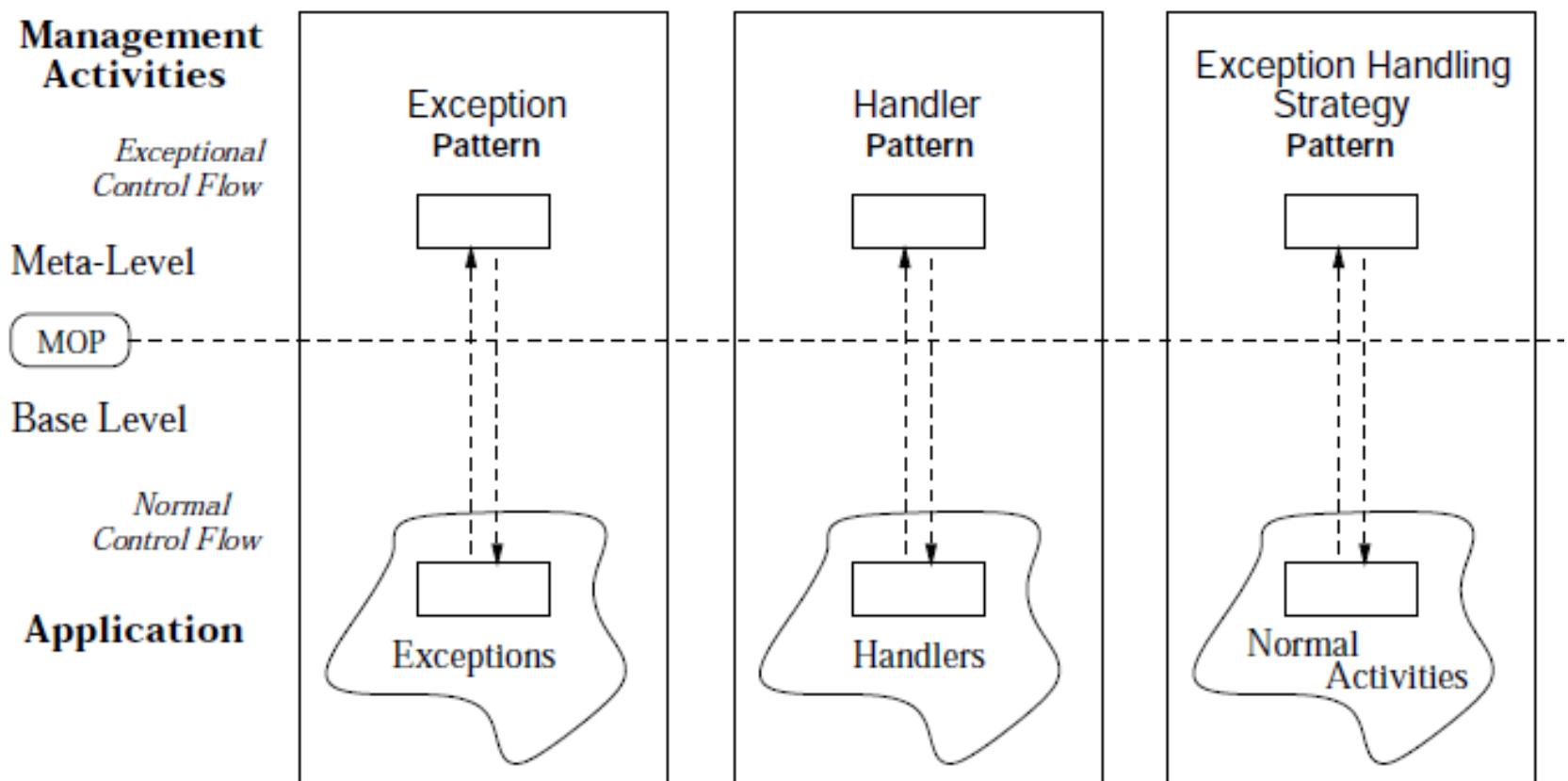
Ação Atômica Coordenada

Ação atômica	Transação Atômica	Exceções
Atividades Colaborativas	Competição no acesso a recursos compartilhados	Tratamento (Resolução) de Exceções Concorrentes



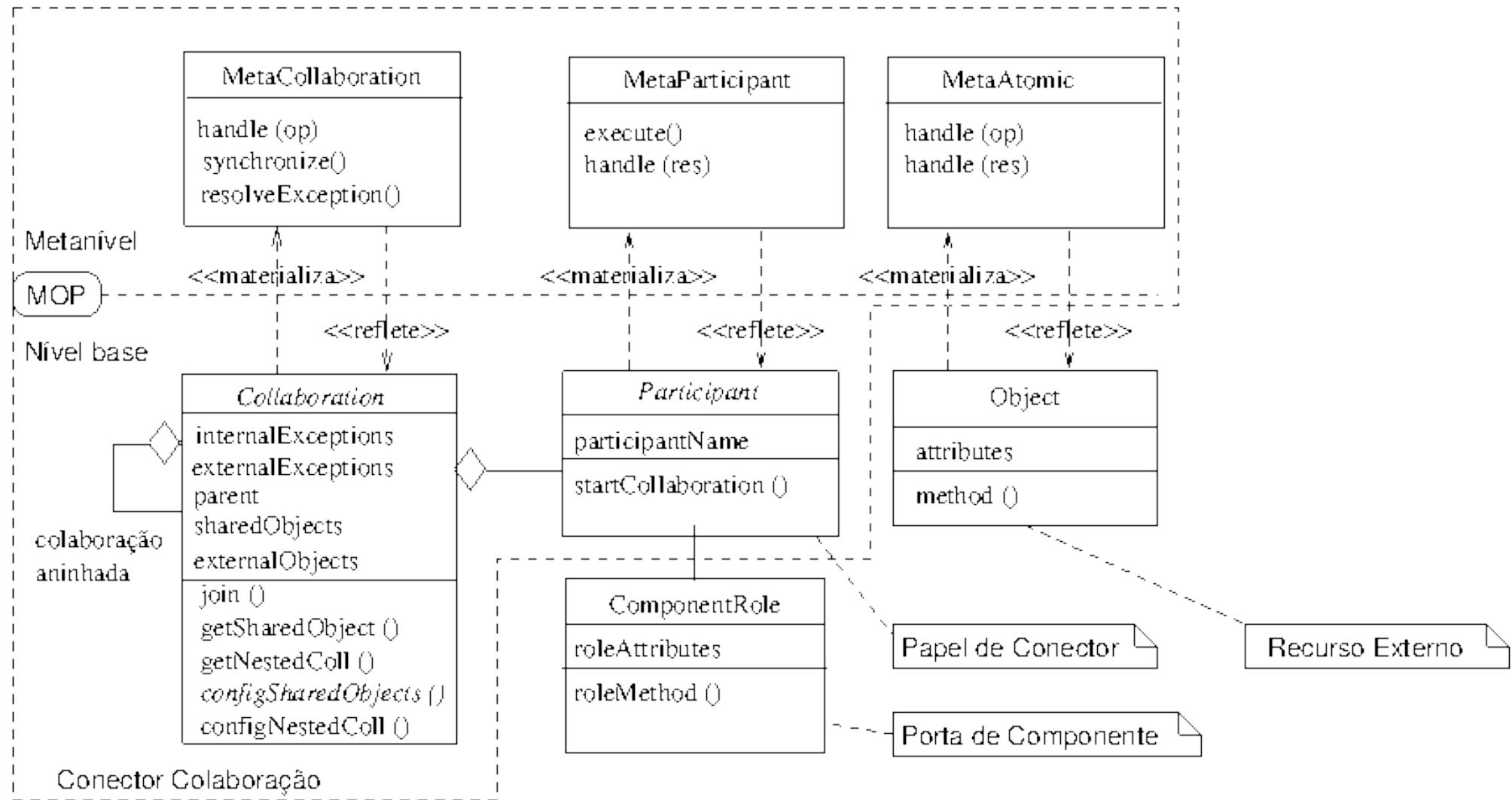


Padrões para Sistemas de Tratamento de Exceções





Arquitetura OO Reflexiva para CA Actions





- 1994: Arche Issarny
- 1995: CA Actions
- 1999: DMI Zorzo
- 2003: Tartanoglu



Considerações Finais



Obrigada pela atenção!

Cecília Mary Fischer Rubira
[cmrubira@ic.unicamp.br]



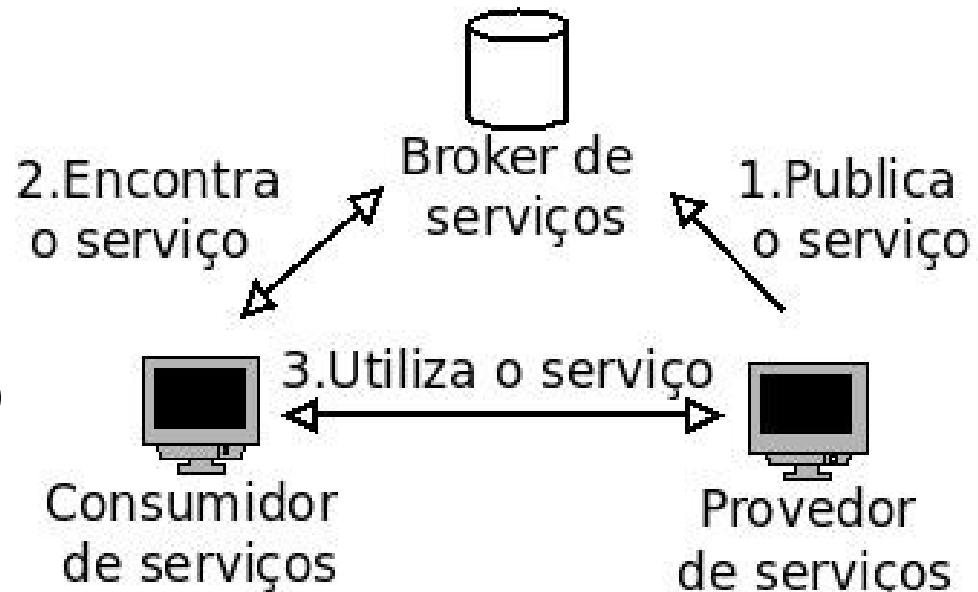


Parte 4: Arquiteturas contemporâneas com tratamento de exceções



Arquiteturas Orientadas a Serviços (I)

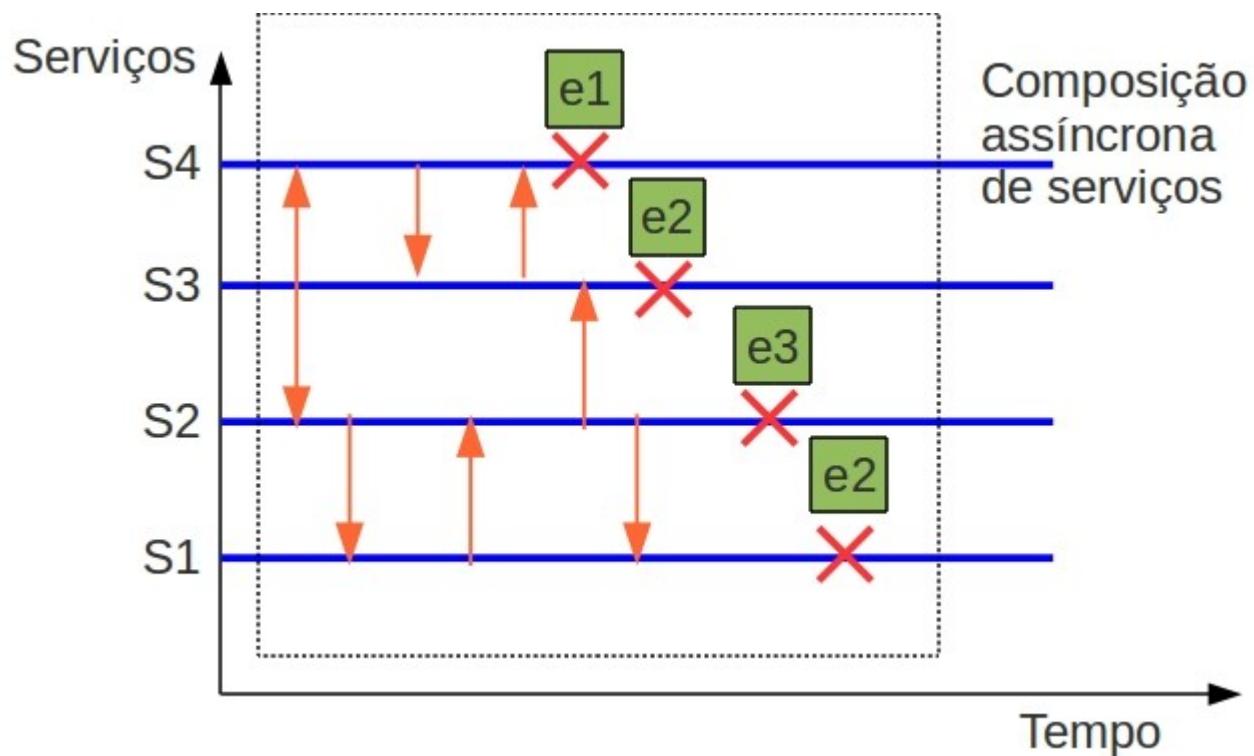
- Aumentam eficiência, agilidade e produtividade de negócios.
- Serviços:
 - Unidade distribuída
 - Auto-contida
 - Composta de especificação e implementação
- Composição de serviços:
 - Serviços síncronos
 - Serviços assíncronos





Arquiteturas Orientadas a Serviços (II)

- Diferentes tipos de exceções podem ser lançadas por diferentes serviços quando um ou mais serviço falha.





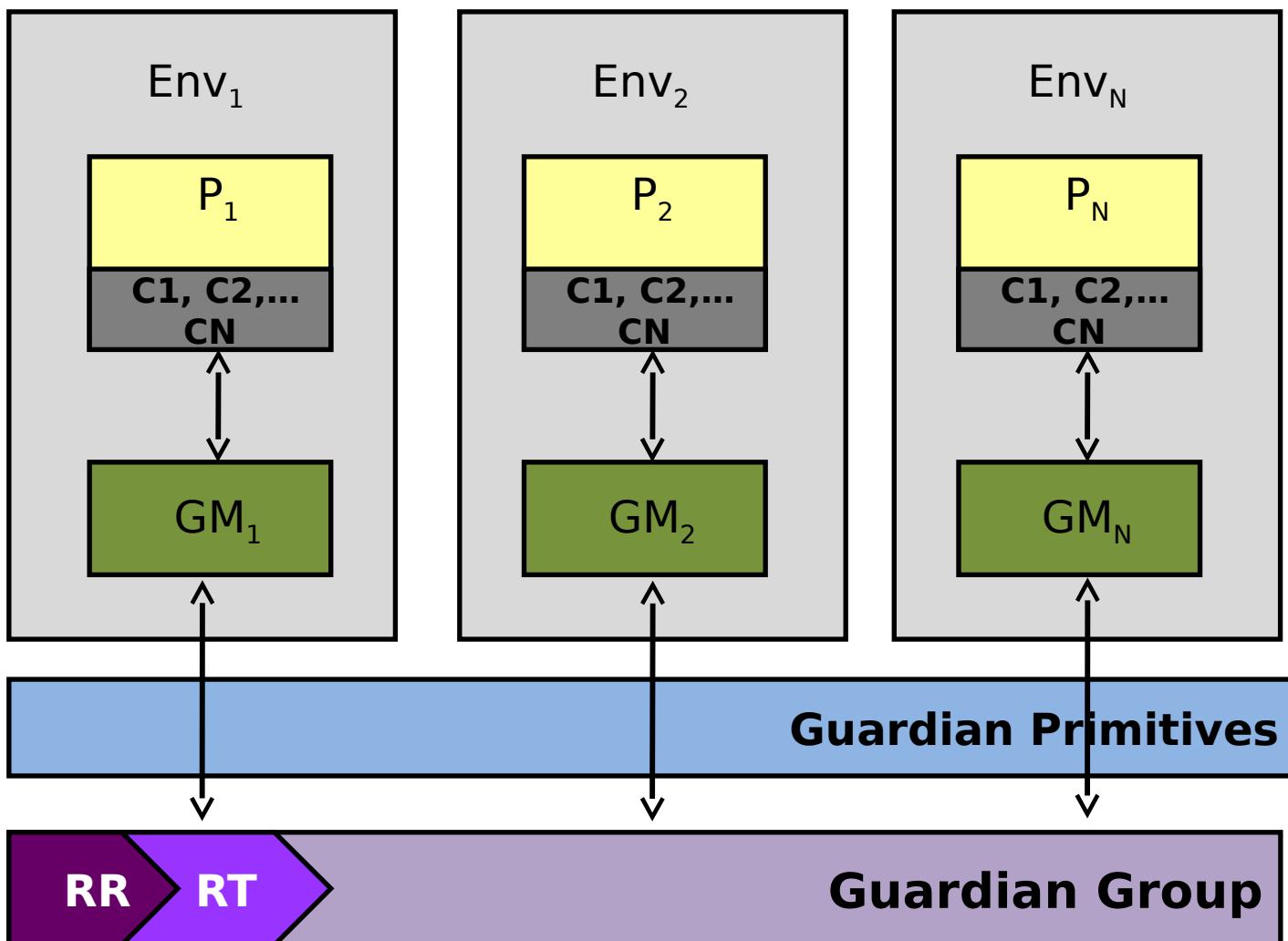
Modelo Guardião

- Um framework conceitual para tratamento de exceções em sistemas distribuídos assíncronos.
- Baseado numa **entidade global** que coordena o tratamento de exceções.
- Pode ser usado para descrever mecanismos de tratamento de exceções.
 - Em particular, aqueles que requerem coordenação (CA Actions).

R. Miller, A. R. Tripathi. The Guardian Model and Primitives for Exception Handling in Distributed Systems. IEEE Transactions on Software Engineering, 30(12), pp. 1008-1022, 2004.



Modelo Guardião





Desenvolvimento de Software Orientado a Aspectos (AOsd) (I)



- **Interesse transversal** (*crosscutting concern*) é um requisito que afeta várias unidades do sistema.
- Exemplos: *logging*, *security*, tratamento de exceções, etc.
- Um **aspecto** é uma unidade modular que entrecorta vários elementos de um sistema.
- Aspectos representam interesses transversais.



Desenvolvimento de Software Orientado a Aspectos (II)

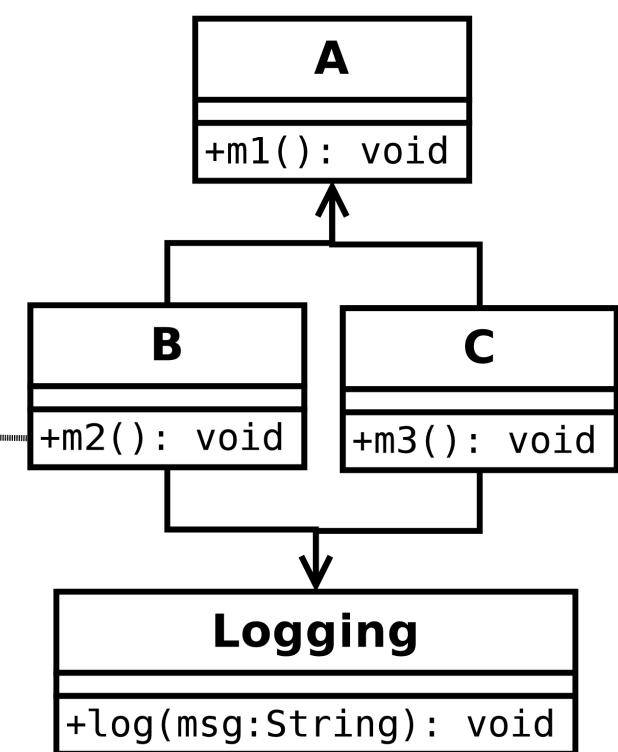


- Aspectos são invocados em pontos de execução específicos, os join points.
- Um *join point* é um ponto de execução identificável pelo sistema (e.g. chamada de método, atribuição de variável)
- *Pointcuts* capturam *join points* no sistema.
- *Advice* implementa o comportamento nos *join points* capturados por um *pointcut*.

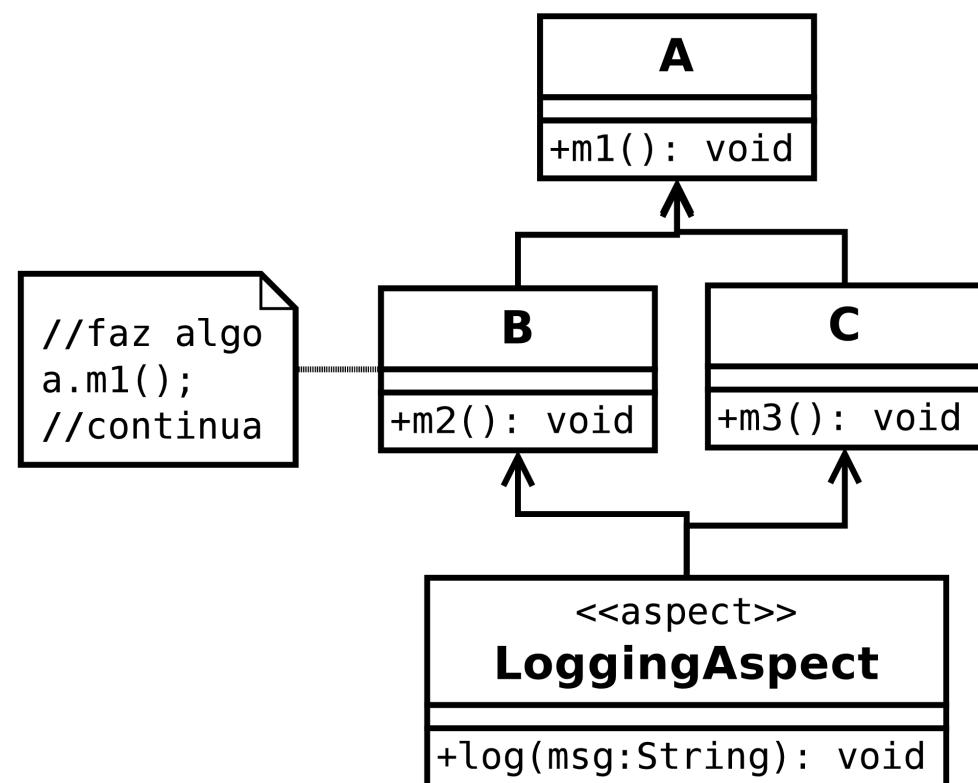


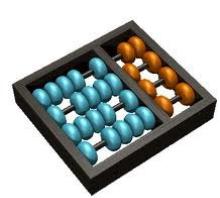
Sem aspecto

```
//faz algo  
a.m1();  
logger.log("Método m1() foi executado");  
//continua...
```



Com aspecto





Exemplo

```
//isto é um aspecto

public aspect LoggingAspect {

    //isto é um pointcut chamado logging
    public pointcut logging():call(void A.m1());

    //isto é um advice
    after():logging(){
        this.log("Metodo m1() foi executado");
    }

    private void log(String msg){
        // realiza o log
    }
}
```



Aspectos e comportamento excepcional

- O tratamento de exceções pode ser modelado como um interesse transversal.
- Aspectos são usados para implementar os tratadores de exceções.
- Extração de blocos try-catch para os aspectos.
- Diretrizes para extrair o comportamento excepcional.



Tratadores como aspectos

```
// original code  
class C {  
    void m() {  
        try{doSomething();}  
  
        catch(E e) { ... }  
    }  
}
```

```
// refactored code  
class C {  
    void m() {  
        doSomething();  
  
    }  
  
aspect A {  
    pointcut pcd :  
        execution(void C.m());  
    void around() : pcd() {  
        try { proceed(); }  
        catch(E e) { ... }  
    }  
    declare soft : E : pcd();  
}
```



Catálogo de refatoração

Scenario	Tangled try-catch block		Nested try-catch block		Terminal exception-throwing code		Handler depends on local vars.		Flow of control after handler execution				Should extract?		
	yes	no	yes	no	yes	no	read	write	no	m	p	r	le	lc	
Untangled Handler	X		X	X	X	X			X	X	X	X		0–1	Yes
Tangled, Non-Mask. Handler	X			X	X	X			X	X	X	X		0	Yes
Nested, Non-Mask. Handler	X			X		X	X		X	X	X	X		1	Yes
Tangled Handler, Term. ETC	X				X	X				X	X			0	Yes
Nested Handler, Term. ETC	X			X		X				X	X			1	Yes
Block Handler	X			X	X	X	X		X	X				2–3	Depends
Loop Esc. Handler	X			X	X	X	X		X		X			2–3	Depends
Loop Cont. Handler	X			X	X	X	X		X			X		2–3	Depends
Context-Dependent Handler	X	X		X	X	X	X	X	X	X	X	X	X	2–4	Depends
Nested, Context-Dependent Handler	X	X	X		X	X	X	X	X	X	X	X	X	3–5	No
Context-Affecting Handler	X	X	X	X	X	X	X	X	X	X	X	X	X	3–8	No

F.Castor Filho, N.Cacho, E.Figueiredo, R.A.M.Ferreira, A.F. Garcia & C.M.F.Rubira. Exceptions and Aspects: The Devil is in the Details. FSE'2006

F.Castor-Filho, N.Cacho, E.Figueiredo, A.F.Garcia,C.M.F.Rubira, J.S.Amorim, H.O.Silva. On the Modularization and Reuse of Exception Handling with Aspects. SPE, 2009



Linhas de Produto de Software (LPS)

- É composta por produtos similares entre si, ou seja, possuem características **comuns** e **variáveis**
- Produtos são gerados a partir dos mesmos ativos centrais
- **Ativos centrais** são artefatos de software reutilizáveis (e.g. casos de uso, arquiteturas, códigos-fonte)





Exemplo

A Product Line of Restaurant Menu Items



Hamburger



Cheeseburger



Dbl. hamburger



Dbl. cheeseburger

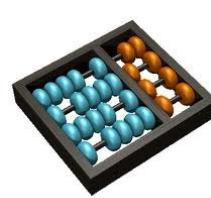


Veggie burger

Do you see the
components, the
architecture, and
the *reuse* in these
products?

Source: www.burgerking.com





Variabilidade de Software

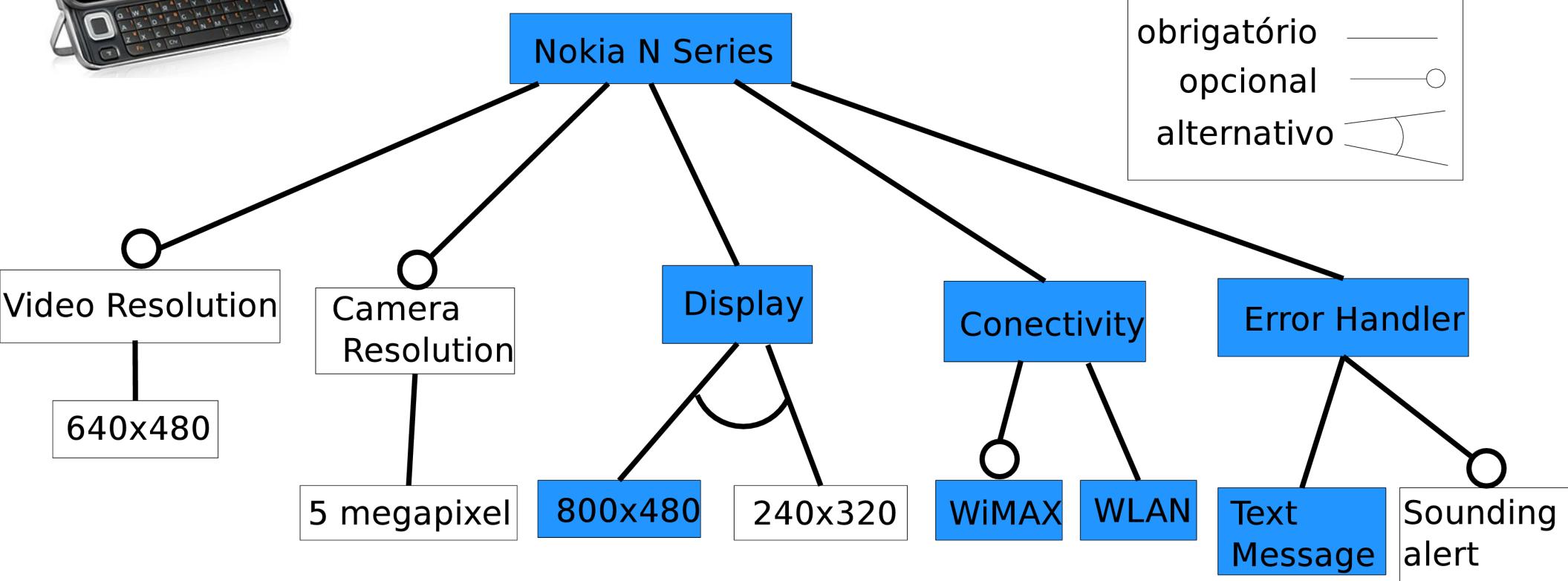
- É a capacidade de um sistema de software de ser modificado, *customizado* e configurado para um específico contexto.
- É uma mudança antecipada, com pontos de variação bem definidos.
- Ortogonal ao desenvolvimento de uma linha de produto.



Linhas de Produto de Software



N810 WiMAX



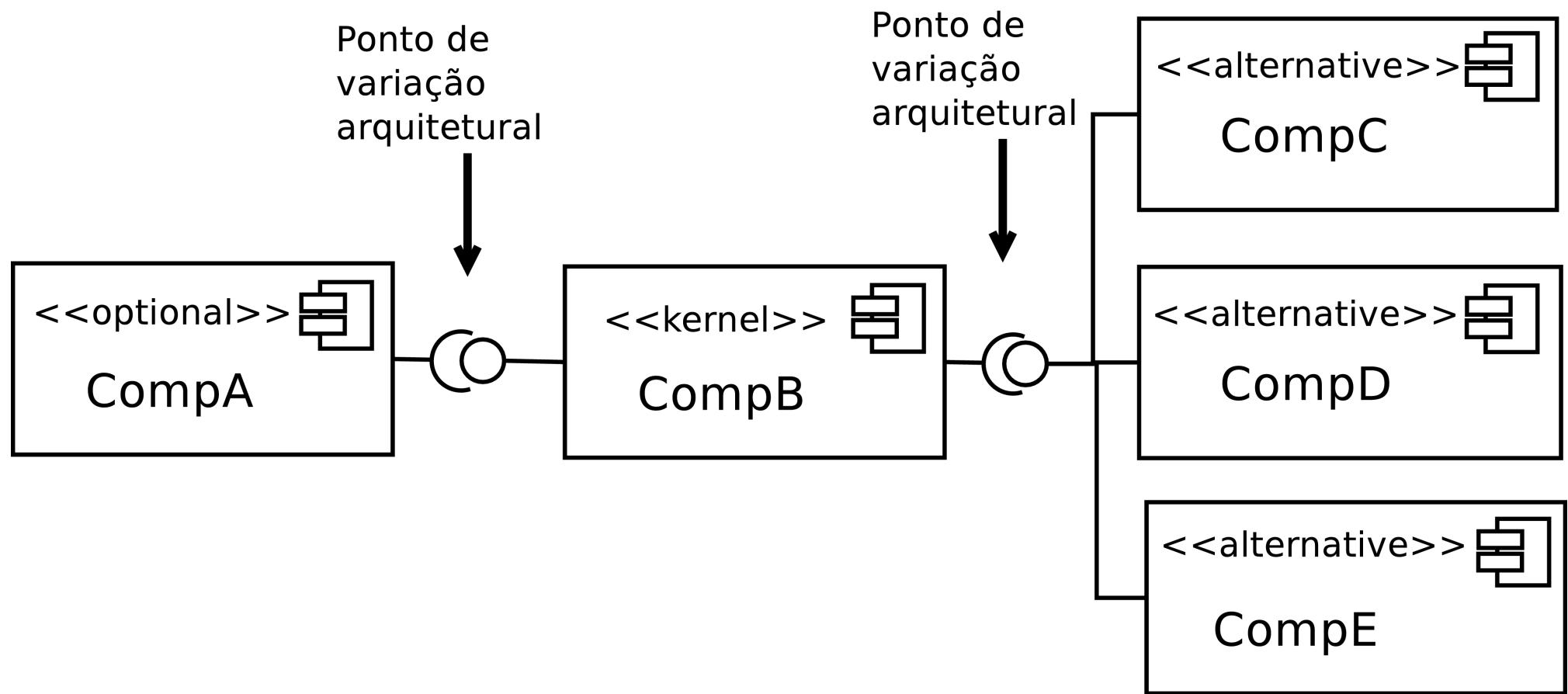


Arquitetura de Linha de Produto (I)

- Oferece explicitamente mecanismos de variação para apoiar a diversidade de produtos.
- Composta por componentes e conectores que podem ser obrigatórios (*kernel*), opcionais (*optionals*) ou alternativos (*alternatives*).
- **Pontos de variação arquiteturais** são os pontos da arquitetura que implementam os mecanismos de variação.

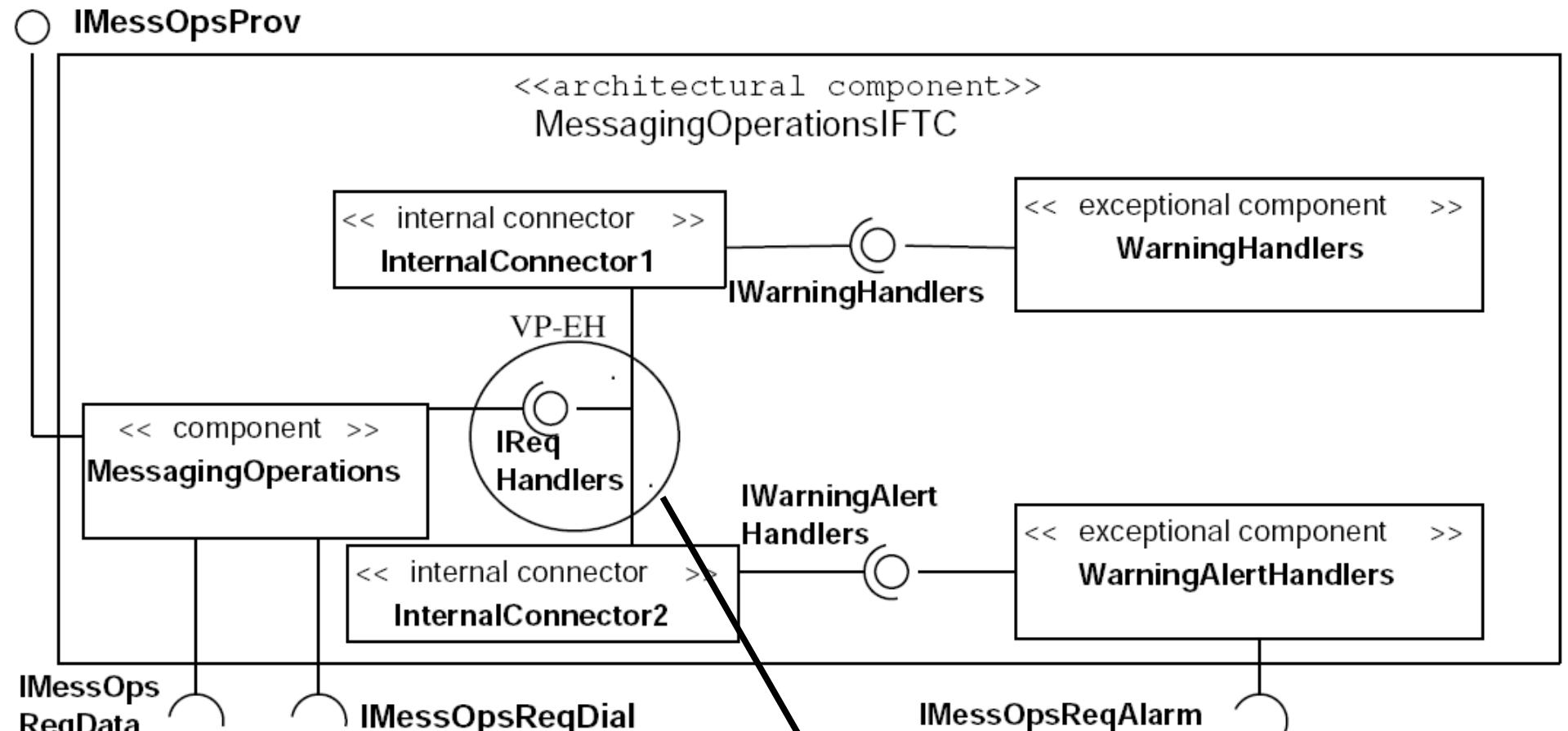


Arquitetura de Linha de Produto (II)





Componente TF baseado em tratamento de exceções

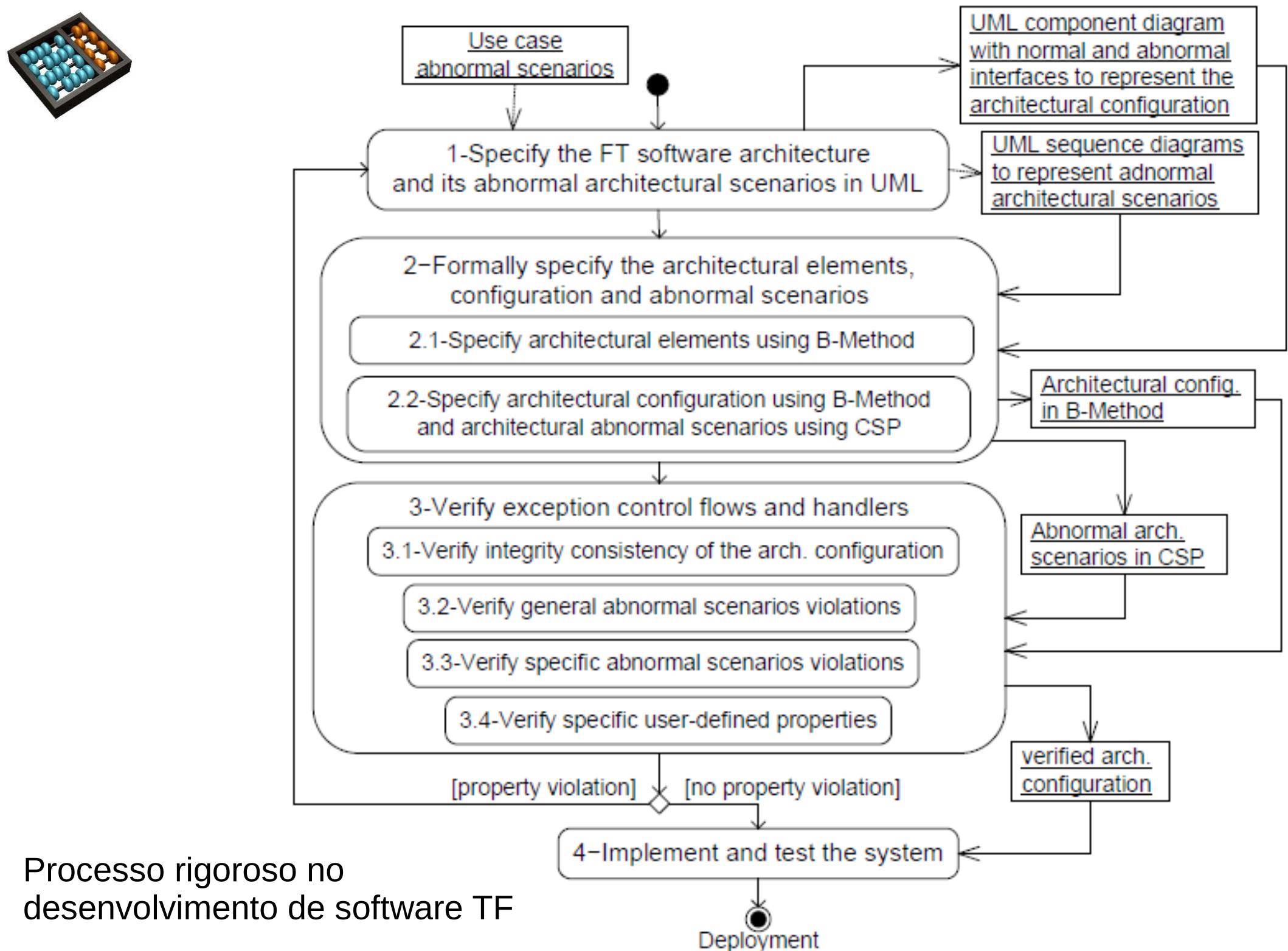


Variabilidade no tratamento de exceções:
diferentes tratadores dependem da seleção de características opcionais/alternativas



Bibliografia

- (1) R. Charette. Why software fails. IEEE Spectrum, 2005.
- (2) R. Charette. This car runs on code. IEEE Spectrum, 2009.
- (3) J. Jézéquel & B. Meyer. Design by contract: the lessons of Ariane. Computer, vol.30, no.1, 1997.
- (4) P.A. Lee e T. Anderson. Fault Tolerance - Principles and Practice. Springer-Verlag 2nd revised edition, 1990.
- (5) I. Sommerville. Software Engineering. Addison-Wesley, 6th edition, 2001.
- (6) R. Wirfs-Brooks. Towards Exception-Handling Best Practices and Patterns. IEEE Software, vol.23, No. 5, 2006.



Processo rigoroso no
desenvolvimento de software TF