




Laboratório de Engenharia de Software

Teste Automatizado 3

Arndt von Staa
Departamento de Informática
PUC-Rio
Maio 2015

Especificação



Laboratório de Engenharia de Software

- Objetivo desse módulo
 - propor soluções para a *manutenção dirigida por testes*
- Justificativa
 - mais de 70% do esforço despendido com criação e alteração de software é gasto em manutenção
 - "se um software é utilizado, ele será alterado", ou seja, passa por sucessivas evoluções ou correções
 - entretanto manutenção tende a desestruturar os sistemas
 - empresas maduras possuem um número grande de sistemas que foram desenvolvidos e mantidos por vários anos sem obedecer às boas práticas atuais
 - sistemas legados são de maneira geral mal estruturados
 - é interessante então dispor de um processo de teste automatizado que apoie a manutenção
 - facilita a refatoração durante a manutenção


Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

2

Laboratório de Engenharia de Software

Software longo




- Enquanto organizações perdurarem
 - sistemas essenciais a seus negócios perdurarão também
 - espera-se que organizações perdurem por muitos anos
 - diversos deles não podem ser interrompidos
 - nem para manutenção planejada
 - muito menos devido a falhas ou devido a agressões bem sucedidas
 - exemplos
 - grid elétrica: comando e controle do despacho de carga
 - aviação: controle de tráfego aéreo
 - refinarias: sistema de comando e controle da refinaria
 - bancos: contas correntes
 - comércio eletrônico: sistema interativo de venda
 - busca eletrônica: sistema interativo de consulta

Mai 2015
Arndt von Staa © LES/DI/PUC-Rio
3

Laboratório de Engenharia de Software

O problema da longevidade



- Podemos prever como o sistema que estamos desenvolvendo agora será daqui a cinco anos?
 - isso é um exercício vão?
 - existem opiniões dizendo que sistemas são substituídos entre 10 a 20 anos
 - os escritos em COBOL vivem mais...
 - mais de 200 bilhões de linhas de código COBOL estão em operação no mundo
 - por dia centenas de vezes mais transações são processadas por COBOL do que transações Google

- COBOL – continuing to drive value in the 21st century, DATAMONITOR, 2008
- Atwood, J.; COBOL: Everywhere and Nowhere; 2009; Acessado 31/out/2013; URL: <http://www.codinghorror.com/blog/2009/08/cobol-everywhere-and-nowhere.html>

Mai 2015
Arndt von Staa © LES/DI/PUC-Rio
4

O problema da longevidade



- Ao invés de se preocupar com daqui a n anos, por que não se preocupar com desenvolver e manter **sistemas continuamente manuteníveis**?
- A tendência atual é sistemas virem a se comunicar com outros
 - possivelmente em organizações diferentes
 - então é importante que sistemas sejam arquitetados, projetados e desenvolvidos para serem **integráveis** e/ou viabilizar a **interoperabilidade**
- Mas ..., não devemos adicionar características que "poderiam ser úteis", o que se quer é
 - uma boa organização interna, ex. modularidade
 - poder reconstruir e retestar com rapidez e facilidade


Por que manter?



- Um artefato em uso (aceito) pode ser alterado para
 - **satisfazer novos** requisitos, regras de negócio, leis, plataformas
 - **corrigir defeitos** identificados após a aceitação
 - +- 70% das falhas devem-se a especificações defeituosas
 - **prevenir** custos de manutenção futura
- Processo de desenvolvimento incremental
 - um artefato aceito em um incremento anterior **pode necessitar de alterações** para poder ser incorporado ao construto correspondente a um novo incremento
 - integrar com outro sistema é uma forma de incrementar

Laboratório de Engenharia de Software

Por que manter?



- +- 70% do custo do software deve-se à sua manutenção
 - Evolução (melhorias)
 - Linux 12% Sis inf 65%
 - Adaptação
 - Linux -- Sis inf 18%
 - Correção
 - Linux 87% Sis inf 17%
 - Outros
 - Linux 1% Sis inf -


• *Linux*: Schach, S.; Jin, B.; Yu, L.; Heller, G.Z.; "Determining the Distribution of Maintenance Categories: Survey versus Measurement"; *Empirical Software Engineering* 8; Kluwer; 2003; pp 351-365

• *Sis inf*: Nosek, J.T.; Palvia, P.; "Software Maintenance Management: changes in the last decade"; *Software Maintenance: Research and Practice* 2(3); 1990; pp 157-174

Mai 2015
Arndt von Staa © LES/DI/PUC-Rio
7

Laboratório de Engenharia de Software

Não existe software perfeito



- +- 50% software entregue contém defeitos não triviais [Boehm]
- Perfeição não existe, densidade de defeitos [Bao]:
 - INTEL: no more than 80-90 defects in Pentium
 - Standard Software: 25 defects / 1,000 lines of delivered code (kLOC)
 - Good Software: 2 defects / 1,000 lines
 - Space Shuttle Software: < 1 defect / 10,000 lines
 - Cellular Phone: 3 defects / 1,000 lines
- Na realidade isso é uma verdade parcial, existem componentes corretos
 - provados estarem corretos usando técnicas especializadas por equipes altamente especializadas

Boehm, B.W.; Basili, V.R.; "Software Defect Reduction Top 10 List"; IEEE Computer 34(1); Los Alamitos, CA: IEEE Computer Society; 2001; pags 135-137

Bao, Xinlong; *Software Engineering Failures: A Survey*; School of EECS, Oregon State University, Corvallis, OR;

Woodcock, J.; Larsen, P.G.; Bicarregui, J.; Fitzgerald, J.; "Formal Methods: Practice and Experience"; *ACM Computing Surveys* 14(4); 2009; pp 1-40

8
Mai 2015
Arndt von Staa

Não existe software perfeito



- Por que podem existir defeitos remanescentes após cuidadoso controle da qualidade de cada módulo?
 - a geração de um erro pode depender do caminho percorrido, considerando a totalidade de módulos,
 - falhas podem ser observadas ao integrar **artefatos (supostamente) corretos**, i.e. artefatos aceitos
 - quase sempre é inviável uma suíte de teste percorrer todos os caminhos possíveis no conjunto de todos os módulos
 - mesmo quando se considera $\text{arrasto} = 1$
 - a especificação pode estar errada: solução correta do problema errado...

Não existe **teste** perfeito



- Suítes de teste **podem ser problemáticas**
 - baixa eficácia
 - eficácia :: $\frac{\text{<número de defeitos encontrados>}}{\text{<número de defeitos existentes>}}$
 - $\text{<número de defeitos existentes>}$ não é conhecido
 - aproximação: eficácia :: $\frac{\text{<número de mutantes mortos>}}{\text{<número de mutantes inseridos>}} - \frac{\text{<número de mutantes equivalentes>}}{\text{<número de mutantes inseridos>}}$
 - não testam de modo a observar possíveis falhas relevantes existentes
 - consequência: defeitos permanecem e falhas poderão ser observadas em tempo de uso
 - falta de sensibilidade
 - suítes produzem evidências de erros, mas os oráculos usados não observam essas evidências

Não existe teste perfeito

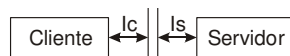


- Suítes de teste tendem a estar **incompletas**
 - isso é um problema intrínseco aos testes
 - a escolha de dados de teste pode influenciar os resultados
 - a explosão combinatória impede testes completos
 - "O planejamento de testes é governado pela necessidade de selecionar **alguns poucos casos de teste** de um gigantesco conjunto de possíveis casos
 - Independentemente de quão cuidadoso você seja, **você deixará de incluir alguns casos relevantes**
 - Independentemente da perfeição do seu trabalho, **você nunca encontrará o último defeito** e, se encontrar, **jamaís o saberá**" [Kaner, C.; Falk, J.; Nguyen, H.Q.]
 - permitem que artefatos **supostamente** corretos contenham **defeitos remanescentes**
 - maior "eficiência" no desenvolvimento às custas de menor eficácia
 - o problema é exacerbado quando suítes forem criadas sem obedecer a adequados critérios de seleção de casos de teste

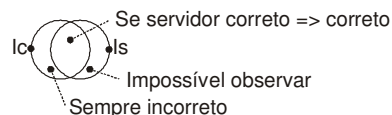
Não existe teste perfeito



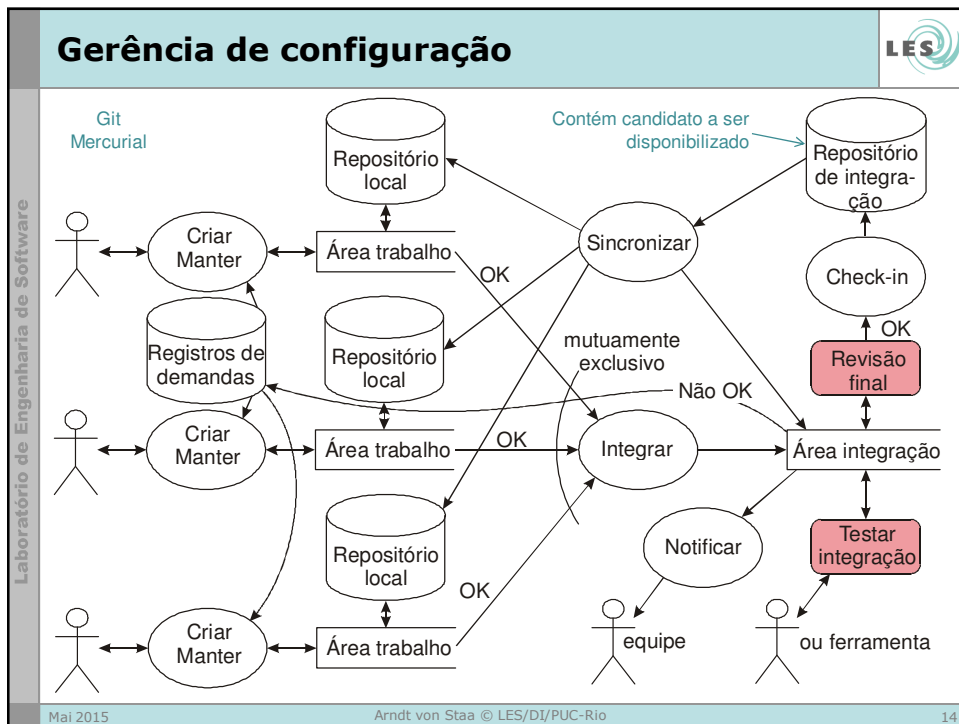
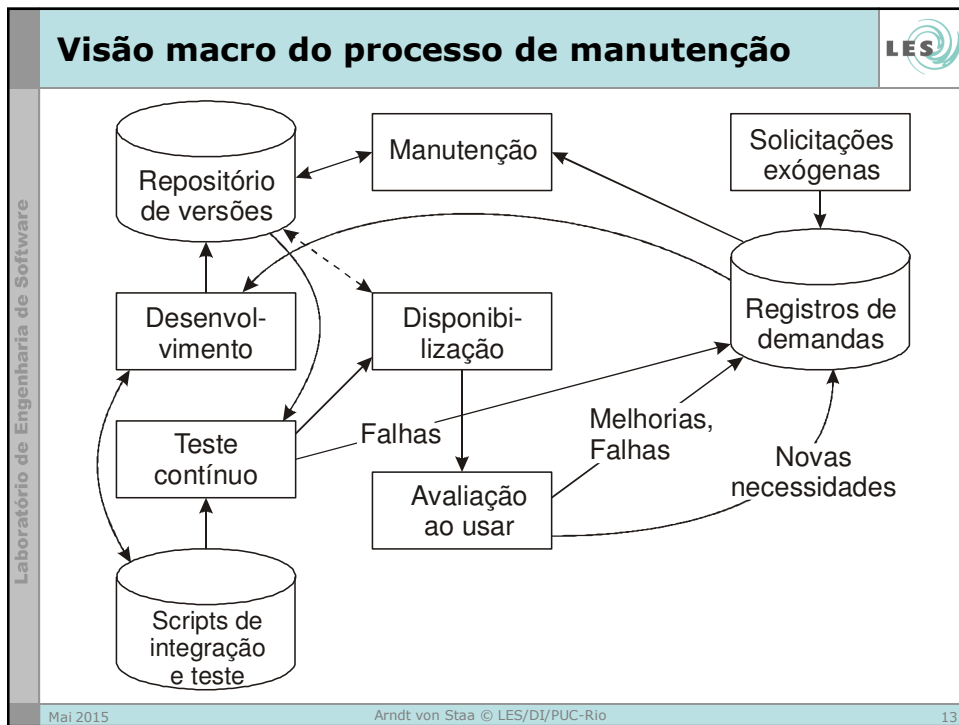
Ao testar interfaces entre artefatos

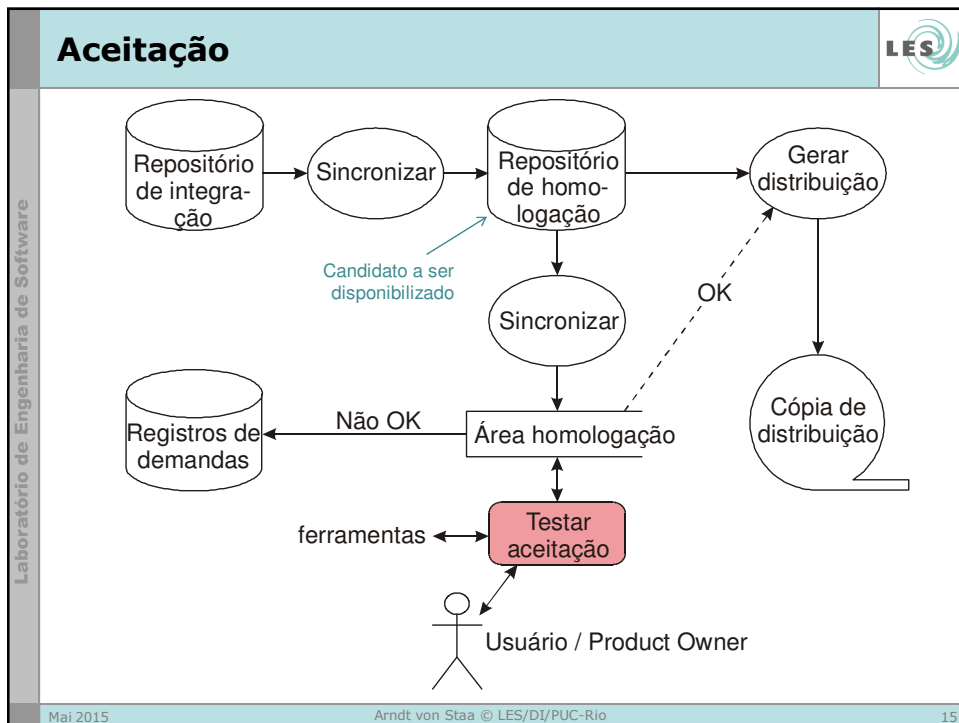


Ic - interface do ponto de vista do cliente
Is - interface do ponto de vista do servidor



- **Consequência:**
 - a interface deve ser única para cliente e para servidor
 - alteração da interface deve implicar a alteração dos dois
 - a interface deve estar especificada de forma completa
→ interface conceitual
- **Interface conceitual**
 - parâmetros, variáveis membro de objeto, variáveis globais, exceções, ...
- **Codificação**
 - ex. string[20], double
- **Propriedades semânticas**
 - métrica
 - ex. nome; distância
 - escala
 - ex. abreviado segundo critério z; metro
 - precisão
 - ex. 40 char; mm
- **Estado das estruturas**





Desenvolva e teste para poder manter

- Consequência: as suítes de teste devem fazer parte do **subsistema de manutenção**
 - viabiliza a manutenção de todos os artefatos: módulos, componentes, sistema, suítes de teste, documentação, ...
 - viabiliza o teste de regressão de módulos, componentes e sistema
 - realiza testes (**quase**) sem necessitar de dublês
 - possivelmente necessita de alguns "mock", exemplos:
 - banco de dados para teste
 - simulação de características do sistema, exemplos:
 - geração de erros de plataforma para avaliar a capacidade e correteude do tratamento dos problemas observados
 - eliminação de esperas ao realizar testes de sistemas de controle
 - calibração de sensores, atuadores e computação

Maio 2015

Arndt von Staa © LES/DI/PUC-Rio

16

Desenvolva e teste para poder manter



- Quando termina o desenvolvimento incremental, os módulos dublê terão sido substituídos por versões de produção
 - criar um ambiente de teste para artefatos compostos
 - componentes, bibliotecas
 - programas, sistemas
 - pode-se gerar construtos visando testes de módulos específicos em que cada módulo sob teste é testado dentro do contexto dos demais módulos
 - objetivo: permitir a evolução sem prejudicar a qualidade dos testes
 - teste de regressão
 - o módulo sob teste é precedido de um módulo de teste específico que faz parte do arcabouço de teste
 - a suíte de teste de manutenção precisa, possivelmente, ser adaptada a partir da suíte usada ao desenvolver
 - não depender mais de dublês

Categorias de manutenção



- Tradicionais
 - manutenção **corretiva**
 - visa eliminar defeitos
 - de arquitetura, projeto, código, ...
 - visa melhorar requisitos não funcionais:
 - usabilidade, desempenho, aparência, ...
 - manutenção **perfectiva** ou **evolutiva**
 - visa atender a novas necessidades
 - integrar com outros sistemas
 - ...
 - manutenção **adaptativa**
 - visa adaptar a mudanças contextuais
 - do substrato, da legislação, das práticas empresariais, ...

substrato: plataforma: hardware, sistema operacional, ...; serviços de terceiros: rede, base de dados, nuvem, ...; tecnologia de desenvolvimento, ...

Categorias de manutenção



- Não tradicionais
 - manutenção **preventiva**
 - visa eliminar dívidas técnicas
 - caracterização: funciona, mas oferece riscos significativos para mantê-lo
 - tal como juros
 - » dívida técnica custa
 - » quanto mais tempo persistir, maior o custo acumulado
 - engenharia **reversa**
 - **paleontologia de sistemas legados** – descobrir qual a sua arquitetura, seu projeto, como e por que funciona
 - **reengenharia**
 - “**refactoring**” da arquitetura e/ou dos projetos dos componentes
 - **rejuvenescimento**
 - visa substituir uma solução senil por outra adaptada às possibilidades modernas, **sem usar “wrappers”**

Paleontologia: do grego *palaiós*, antigo + *óntos*, ser + *lógos*, estudo

Problema da manutenção: *co-evolução*



- Desenvolvimento incremental ou manutenção sucessiva implicam a necessidade de **co-evolução**
 - a **cada evento** de evolução ou correção de um artefato
 - o artefato, a suíte de teste, a documentação, o projeto, a arquitetura, a especificação, entre outros, precisam voltar a estar (quase) **coerentes entre si**
 - a co-evolução implica a repetição dos testes
 - sempre que um artefato for alterado
 - a **evidência do teste** realizado precisa estar em acordo com as alterações
 - pode tornar necessária a repetição dos testes de **artefatos adjacentes** (clientes ou servidores) do artefato alterado
 - mesmo quando for realizado de forma manual
 - **co-evolução pode induzir custos elevados**
 - *hipótese*: quanto mais modular, menos artefatos precisarão ser co-evoluídos, portanto menor será o custo da alteração

Problema da manutenção : *refatoração*



- À medida que vão sendo feitas alterações em um artefato composto por vários módulos ou componentes
 - **regras** de arquitetura e/ou de projeto são frequentemente **violadas**
 - ex. comunicação direta entre camadas ao invés de respeitar as interfaces entre camadas
 - a consequência é a **deterioração arquitetural**
 - aumenta a frequência de defeitos remanescentes
 - aumenta a dificuldade de manter
 - aumenta custos de manutenção e de danos decorrentes de falhas
 - para **evitar a deterioração** torna-se necessário
 - reorganizar o código (refatoração de baixo nível)
 - algumas vezes **reengenheirar** (refatoração de alto nível)
 - atualizar a arquitetura ou o projeto e co-evoluir o código e a documentação para que volte a estar coerente
 - mudar a estrutura, mas não mudar a funcionalidade

Algumas das atividades de manutenção 1/2



- Registrar **falhas, incidentes** e **solicitações**
- Especificar a manutenção a ser feita
- Diagnosticar
 - localizar os artefatos a serem alterados
- Reengenheirar os artefatos afetados e, possivelmente, seus vizinhos
 - pode envolver a reorganização da arquitetura ou dos projetos
- Criar, alterar, coevoluir, ou excluir os artefatos afetados
 - pode envolver a reescrita de partes substantivas em virtude da mudança do substrato
 - recompilação automatizada de todos artefatos
 - integração automatizada de todos artefatos

scrum, kanban

Algumas das atividades de manutenção 2/2



- Retestar tudo inclusive a documentação
 - teste de regressão
 - teste do que foi evoluído
 - se necessário, homologação, ou certificação
 - dependendo do processo usado: efetuar uma revisão cuidadosa
- Tornar operacional a nova versão
- Registrar versões
- Gerenciar configuração
- Adquirir, registrar e usar medições e estatísticas
 - objetivo: identificar que artefatos devem ser melhorados de modo a reduzir a frequência de falhas e reduzir o custo da manutenção
 - observar a existência de dívida técnica
- n outras

Manutenção dirigida por testes 1 / 4



- Algumas observações feitas na prática
 - se uma classe **bem desenvolvida** e que possui um teste automatizado precisa ser alterada
 - escreva, ou reveja, o teste automatizado antes de modificar o código, de modo que o teste corresponda à nova especificação
 - altere a classe para que passe pelo teste
 - ajuste a classe para que volte a ter boa organização interna

Manutenção dirigida por testes 2 / 4



- Algumas observações feitas na prática
 - se um conjunto de classes interdependentes e **possuindo boa arquitetura** precisar ser alterada
 - se necessário, ajuste a arquitetura ou projeto do conjunto
 - para cada classe do conjunto e respectivos vizinhos
 - escreva ou reveja antes um teste automatizado que teste a classe (algumas poucas classes) segundo a nova especificação
 - reveja ou implemente a(s) classe(s)
 - ajuste a organização interna da(s) classe(s) assegurando boa organização interna
 - reveja o projeto do conjunto, assegurando bom projeto do conjunto
 - se a mudança do projeto tornar necessário, repita esses passos

Manutenção dirigida por testes 3 / 4



- Algumas observações feitas na prática
 - se uma funcionalidade a ser modificada possui uma **arquitetura ou projeto ruim**
 - **(re)faça a arquitetura / projeto** da funcionalidade
 - escreva ou reveja o teste automatizado visando a funcionalidade modificada
 - para cada classe (algumas poucas classes) da nova arquitetura e respectivos vizinhos
 - escreva ou reveja antes um teste automatizado que teste a(s) classe(s) segundo a nova especificação
 - reveja ou implemente a(s) classe(s)
 - refatore a(s) classe(s) assegurando boa organização interna
 - reveja (refatore) o projeto, assegurando bom projeto do conjunto
 - se a mudança do projeto tornar necessário, repita esses passos

Manutenção dirigida por testes 4 / 4



- Ao aplicar a abordagem a um sistema legado (+-10 anos, Java) de uma empresa, o custo acabou sendo mais baixo do que a técnica convencional de fazer "patches"
 - o número de **falhas recidivas** tornou-se zero
 - no experimento realizado isso foi alcançado em menos de um mês
 - o custo total diminuiu, logo:
 - os clientes ficaram mais felizes
 - a direção da empresa ficou mais feliz

Rosa, O.A.L.; (2011) *Test-Driven Maintenance: uma abordagem para manutenção de sistemas legados*; Dissertação de Mestrado; DI/PUC-Rio

Subsistema de manutenção, conteúdo



- Documentação técnica coerente com o que está implementado
 - arquitetura / projeto
- Código fonte satisfazendo padrões estabelecidos, especialmente
 - instrumentação de apoio ao teste
 - instrumentação de apoio à diagnose
- Scripts / módulos de teste disponíveis e coerentes
 - módulo a módulo
 - componente a componente
 - sistema, requisitos funcionais
 - sistema, requisitos não funcionais
- Scripts de **recompilação versionada** disponíveis e coerentes
- Scripts de **integração contínua** disponíveis e coerentes
- Versões das ferramentas (pacotes instaladores) utilizadas registradas no repositório do projeto
- Registros de demandas
- Repositório de versões

Sistemas novos, ideal



- Ao utilizar as ferramentas e disciplinas modernas teremos idealmente:
 - nem sempre ☹ uma boa arquitetura / projeto
 - caso exista, documentação técnica **quase** coerente com o que está implementado
 - código fonte satisfazendo em grande parte os padrões estabelecidos
 - talvez os scripts / módulos de teste disponíveis e coerentes
 - módulo a módulo
 - sistema, requisitos não funcionais
 - logs de execução dos testes
 - tudo registrado em um repositório de versões
- ➔ o subsistema de manutenção está **parcialmente** disponível

Sistemas legados, usual



- Documentação técnica precária, muitas vezes não confiável
- Código fonte
 - frequentemente desorganizado
 - raras vezes satisfaz padrões de programação consagrados
- Scripts de recompilação (*make, ant, ...*) podem existir
- Scripts de integração contínua em geral não existem
- Scripts / módulos de teste automatizado
 - não existem ou estão defasados
- Podem existir roteiros de teste (manual)
- Podem existir roteiros de teste de regressão
- ➔ Praticamente não existe subsistema de manutenção
- ➔ Nem sempre existe repositório de versões

Bibliografia



- Bao, Xinlong ; *Software Engineering Failures: A Survey*; School of EECS, Oregon State University, Corvallis, OR, U.S.A;
- Freeman, S.; Pryce, N.; (2009) *Growing Object-Oriented Software Guided by Tests*; Upper Saddle River, NJ; Addison-Wesley
- Huckle, T.; *Collection of Software Bugs*;
<http://www5.in.tum.de/~huckle/bugse.html> → Bugs in general; last update November 7, 2011
- Persse, J.; *A Basic Approach to ITIL Service Operation: Setting the Foundation for ITIL V3*; Kindle edition; Atlanta: Tree of Press; 2010
- Rosa, O.A.L.; *Avaliação de test-driven maintenance em sistema legado*; Dissertação de Mestrado, Departamento de Informática, PUC-Rio; 2011
- Skwire, D.; Cline, R.; Skwire, N.; *First Fault Software Problem Solving: A Guide for Engineers, Managers and Users*; Ireland: Opentask; 2009
- Thomas, D.; "The Deplorable State of Class Libraries"; *Journal of Object Technology* 1(1); Zürich, CH: ETH Zürich; 2002; pags 21-27; Buscado em: 28/aug/2006;
URL: http://www.jot.fm/issues/issue_2002_05/column2



FIM