




Laboratório de Engenharia de Software

## Teste Automatizado 2

Arndt von Staa  
Departamento de Informática  
PUC-Rio  
Maio 2015

## Especificação

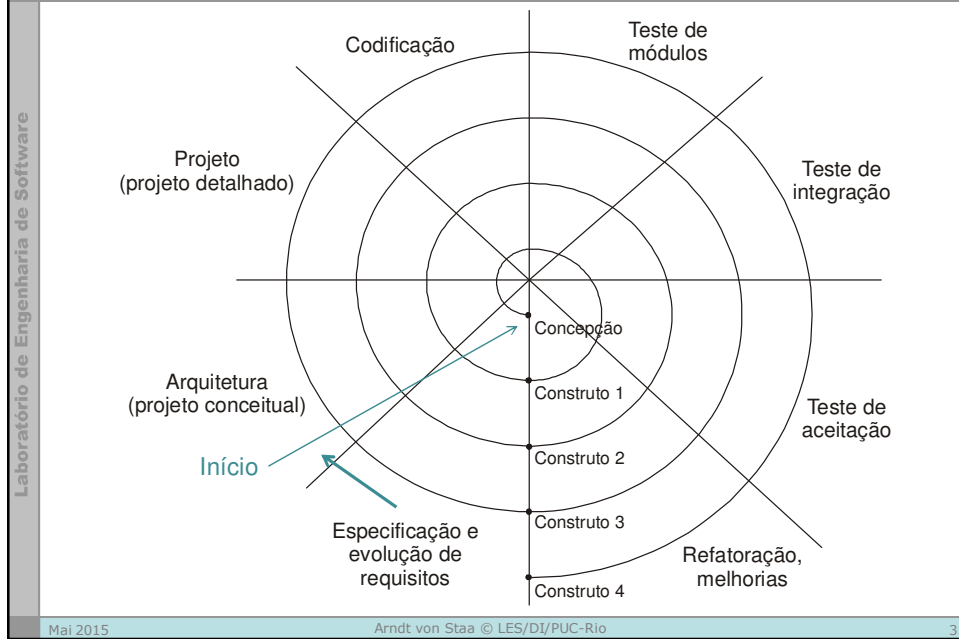


Laboratório de Engenharia de Software

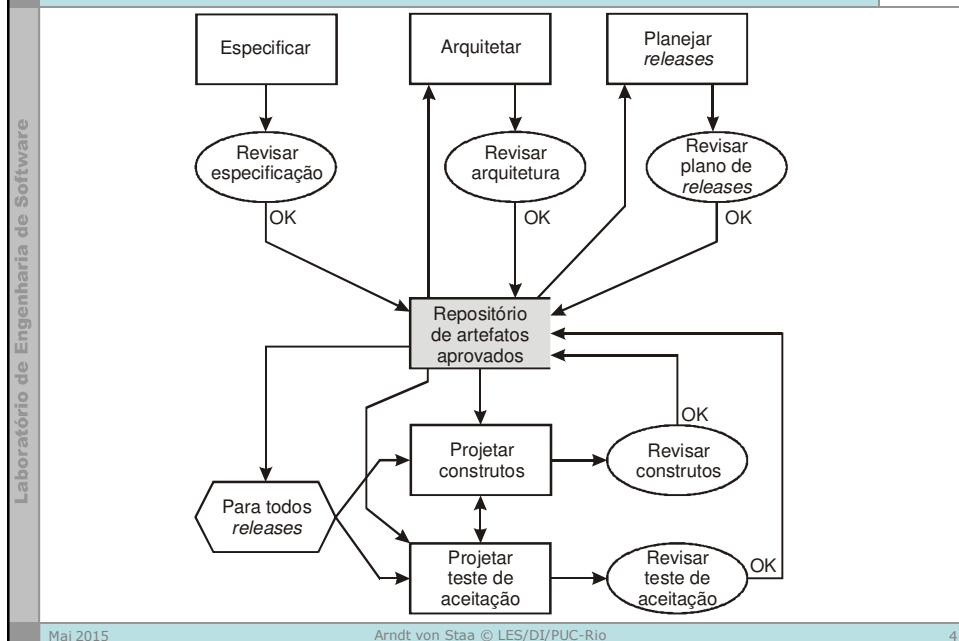
- Objetivo desse módulo
  - discutir o desenvolvimento incremental realizado por equipe
  - apresentar processos de desenvolvimento dirigidos por testes
- Justificativa
  - software é desenvolvido por equipes.
    - problema complexo é assegurar que o trabalho de um membro não conflite com o de outro
  - uma das consequências do teste automatizado é que se pode conduzir o desenvolvimento dirigido por testes
    - primeiro redige-se o teste e depois o módulo objetivo que satisfaz o teste
    - qualquer pessoa ou equipe pode repetir os testes sempre que necessário
  - casos de teste passam a ser uma modalidade de redigir especificações através de exemplos executáveis e verificáveis

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

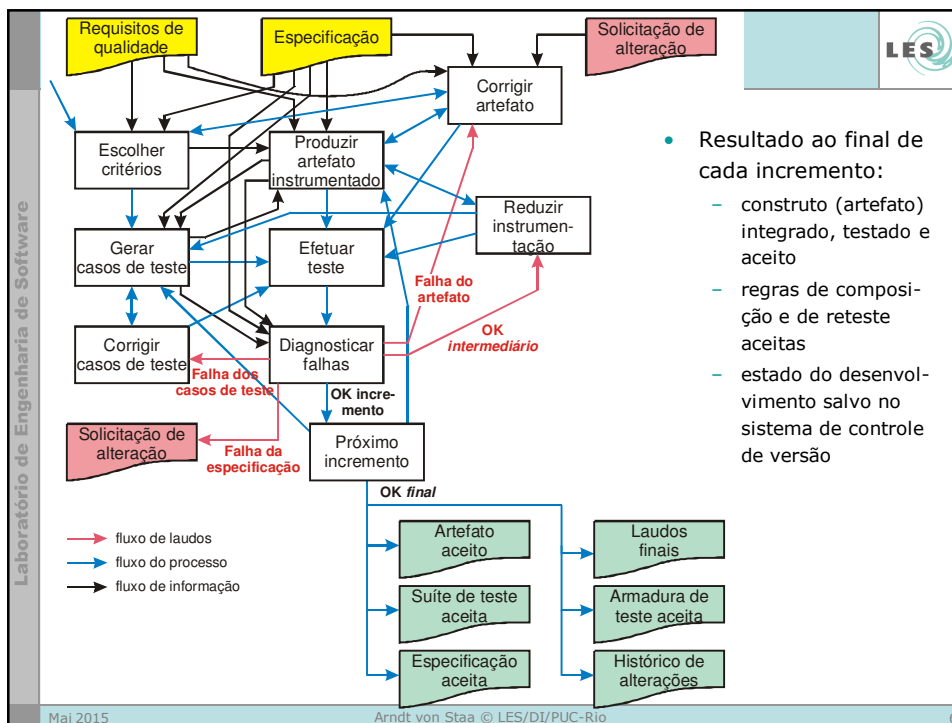
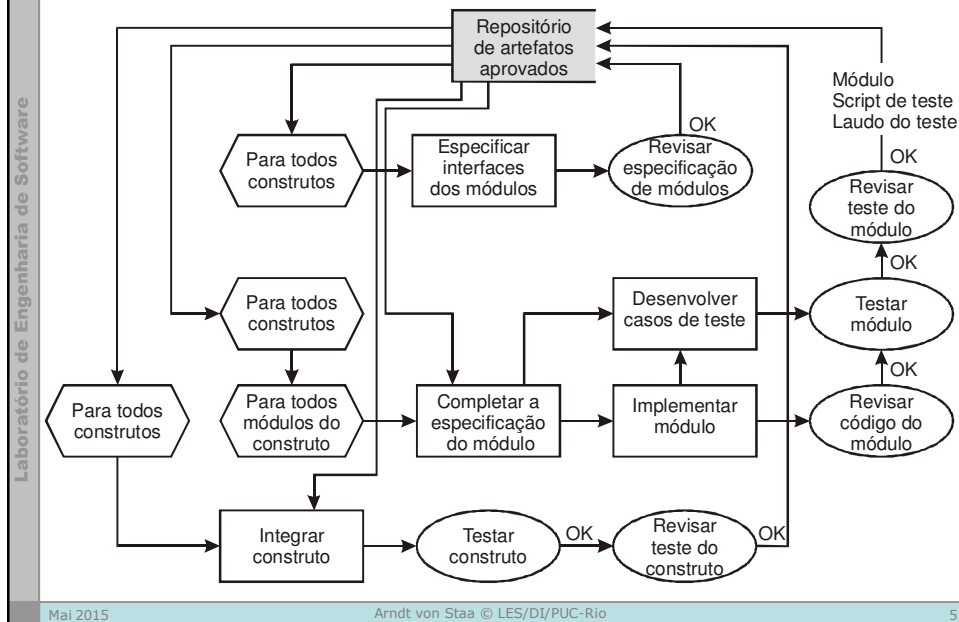
**LES**



**LES**



**LES**



## DDT: Visão macroscópica



- DDT – **desenvolvimento dirigido por testes**
  - TDD – *test driven development*
- Processo – nível alto de abstração
  - Estabelecer contexto → armadura de teste
    - diretórios
    - módulos, bibliotecas, arcabouços, componentes já aceitos
    - composição do construto: **gmake, make, ant, maven, ...**
    - se necessário: bases de dados, arquivos persistentes
  - Especificar um pouco
  - Redigir os casos de teste correspondentes à especificação
  - Testar um pouco → erros por falta de implementação
  - Implementar um pouco, até zero erros de programação
  - Testar a implementação existente até zero falhas
  - Repetir até que o artefato esteja completo
  - Reorganizar, arrumar e retestar o código, **até que**
    - **correto segundo o teste E** satisfaz **qualidade de engenharia**

## Desenvolvimento dirigido por testes



- Especifique a interface (conceitual) do módulo objetivo
  - C, C++ : módulo de definição (*header file*) – **.h, .hpp**
  - Java, diagrama de classes, se conveniente : **interface**
- Ao especificar considere uma das alternativas
  - opção 1: especificação incremental
    - inicia com uma **especificação inicial** (parcial) do módulo objetivo
      - deve ser pequena, mas deve **fazer sentido**
    - à medida que progride o desenvolvimento do módulo objetivo adicionam-se mais requisitos → facilita a adaptação às reais necessidades, quando estas ainda forem nebulosas
    - requisitos também podem ser modificados (corrigidos) de um incremento para outro
  - opção 2: especificação completa
    - comum quando se usa ferramentas de modelagem
    - na prática é **sempre** incremental, porém com uma especificação inicial **quase completa e correta** do **componente** objetivo

## Desenvolvimento dirigido por testes



- Criar o **construto semente** (ou a *release semente*)
  - objetivo deste passo: assegurar que os processos de compilação e de execução dos testes estão operando corretamente
    - usualmente o resultado do teste executado nesta etapa é, usualmente, um volume grande de falhas devidas à **ausência de código**
- Por que “construto semente”?
  - é a partir deste construto que será desenvolvido, de forma incremental, o artefato desejado
    - ver Teste estrutural 3 – armadura de teste

## Desenvolvimento dirigido por testes



- O construto semente é formado pelos artefatos:
  - **contexto** disponível → módulos testados existentes, dublêns
  - scripts de **controle da compilação e do teste**
    - `comp`, `gmake`, `make`, `runTestSuite`, `ant`, `maven`, ...
  - **módulo de definição**, ou **interface**
    - possivelmente parcial
  - **módulo a desenvolver** é um módulo de enchimento simples
  - **módulo de teste específico** do módulo a desenvolver
    - também é um módulo de enchimento simples
  - **script de teste** vazio
    - contém (parte de?) os títulos dos casos de teste criados a partir de algum critério de teste
- Compile e teste até que o **processo esteja operando corretamente** e não existam mais **erros de compilação**
  - podem ser geradas falhas de teste: casos de teste vazios, método de teste específico vazio

## Desenvolvimento dirigido por testes



- O **módulo semente de teste específico** deve conter o controle da interpretação e retornar "*comando não conhecido*" para todos os comandos de teste a interpretar
  - os comandos a interpretar podem ser gerados a partir da interface do módulo a desenvolver (ex. `.hpp`)
- No **módulo semente objetivo**
  - cada **função a ser implementada** na iteração faz nada
    - se a função deve retornar alguma coisa, retorna um valor constante
      - alguma coisa sintaticamente correta mas que provoque erro ao testar
  - as **funções que não serão implementadas** podem, alternativamente, gerar uma exceção: "Método xxx não implementado"
    - sinaliza erros de precedência → métodos implementados que dependem de métodos que não serão implementados

Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

11

## Exemplo



### Construto anterior concluído

```
[Modulos]
str_seg.str    g
string.makeup  g

segment
segabstr
segloc1
segmsg
tst_seg

Main
String

[MacrosApos]
SPECTEST      = tst_seg
```

### Construto novo

```
[Modulos]
str_seg.str    g
str_pg.str     g      cria a tabela de mensagens
string.makeup  g

type_pages_1
pages          define o novo módulo
tst_pg

segment
segabstr
segloc1        usa módulos já existentes
segmsg

Main
String

[MacrosApos]
SPECTEST      = tst_pg  informa o módulo específico de teste
```

Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

12

## Desenvolvimento dirigido por testes



- Selecione a **composição** do incremento a ser implementado
  - determine como será testado o incremento
    - casos de teste adicionados pelo incremento
  - pode requerer
    - a adição de instrumentação ao módulo objetivo
      - assertivas de entrada e saída dos métodos, e estrutural das classes
    - o uso de funcionalidades já implementadas em etapas anteriores
    - a evolução ou correção da especificação da interface do módulo objetivo, e das funcionalidades e casos de teste implementados em incrementos anteriores
  - os casos de teste correspondem a uma especificação definida através de exemplos
    - **hipótese**: dispor dos casos de teste antes de desenvolver reduz os enganos ao desenvolver
    - inicialmente podem ser menos rigorosos → menos tempo de teste
    - à medida que o trabalho progride precisam ser evoluídos para o rigor requerido pelo projeto

## Desenvolvimento dirigido por testes



- Desenvolva o **construto incremento pré-operacional**
  - evolua o módulo de teste específico de modo que possa controlar o teste do incremento do módulo objetivo
    - lê e interpreta os comandos do incremento
  - adicione os *scripts* dos casos de teste, visando o incremento a ser implementado
  - ajuste os scripts de incrementos anteriores de modo que continuem a testar o que já foi implementado, mas agora utilizando o novo incremento
    - se desejado adicione casos de teste ao *script* de teste existente
  - compile e teste
    - o teste reportará falhas por ausência de funcionalidade do incremento do módulo objetivo
    - não deve reportar falhas por falta de interpretador do script
    - não deveria reportar falhas relativas a funcionalidades já existentes

## Desenvolvimento dirigido por testes



- Desenvolva o **construto incremento operacional**
  - objetivo: implementar o incremento do módulo objetivo e assegurar que esteja no nível de qualidade desejado
  - implemente as funções do incremento especificado
    - se necessário reveja o módulo de teste específico e o script de teste
  - compile e teste até que não existam mais falhas e todos requisitos de **qualidade de serviço** estejam aceitos
    - **reveja** os comentários e artefatos complementares (modelos, mensagens, etc.) e
    - reorganize (**refatore**) o código até que todos artefatos estejam coerentes e no nível de **qualidade de engenharia** esperado pelo projeto ou pela organização
      - pode-se saltar esta etapa em alguns incrementos, mas não no último

## Desenvolvimento dirigido por testes



- Ao desenvolver o incremento **adicione novos casos de teste** sempre que julgar que determinada condição ou sequência de execução (caminho) não tenha sido suficientemente testada
  - ideal: use medição de cobertura como apoio
- Selecione as funções do módulo a desenvolver que serão implementadas no próximo incremento
- Repita para o próximo incremento até ter concluído o desenvolvimento
- Antes de dar por concluído o desenvolvimento
  - verifique se a suíte de teste está completa segundo os requisitos de qualidade desejados
    - os critérios de geração selecionados foram completa e corretamente implementados



## Desenvolvimento dirigido por testes



- Resultado:
  - as assertivas e o *script* de teste são na realidade uma **especificação executável e verificável**
    - a especificação é formada por exemplos: casos de teste
      - não é o ideal, mas é eficaz
  - ao terminar o módulo objetivo **estará “correto” segundo a suíte de teste**
    - quanto mais rigoroso for o teste, mais confiável será o módulo
    - o rigor depende da escolha dos casos de teste
      - mais rigor usualmente implica mais casos de teste e custos maiores
    - não esqueça de especificar e testar **requisitos não funcionais**
      - **não satisfazer** a especificação de um requisito não funcional **corresponde** a um defeito, é necessária alguma forma de detectá-lo
  - script de teste
    - quanto mais rigoroso, mais valioso será e mais trabalho dará para ser desenvolvido

## DDT: vantagens




- Exige rigor
  - ao escrever as especificações da interface do artefato
  - ao redigir os *scripts* de teste

**Rigor é sempre vantagem**

- embora alguns não acreditem

Laboratório de Engenharia de Software

## DDT: vantagens



- Facilita o **desenvolvimento e a integração incremental**
  - a cada incremento pode-se retestar integralmente o que já foi feito (teste de regressão)
    - reteste a baixo custo construto a construto na ordem em que os módulos foram acrescentados
    - o que não foi alterado ou acrescentado deve continuar operando tal como esperado
    - o que foi alterado ou acrescentado terá teste específico desenvolvido, completo e operacional
    - nova versão sob controle de versão
  - mesmo quando baseado em incrementos pequenos
    - desenvolvimento iterativo
- Os **testes são repetíveis**
  - encontram repetidamente os mesmos problemas enquanto estes não tiverem sido sanados
  - facilita o diagnóstico e a depuração precisos


Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

19

Laboratório de Engenharia de Software

## DDT: vantagens



- A função de teste específico serve como **exemplo de uso** do módulo
- O *script* de teste serve como **especificação executável e verificável** do módulo objetivo
  - apesar de ser uma especificação incompleta (baseada em exemplos), frequentemente é mais precisa do que as especificações textuais
  - define exatamente o que é esperado através de um grande número de exemplos relevantes
- Facilita corrigir eventuais problemas nas especificações
  - defeitos de especificação são evidenciados cedo

Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

20

## DDT: vantagens



- Reduz o esforço de teste ao levar-se em conta o custo de reteste após corrigir ou evoluir o módulo
  - quanto maior o número de retestes **necessários** maior será a economia
- Reduz o estresse do desenvolvedor
  - permite particionar o desenvolvimento em tarefas com duração de menos de um dia
  - cada tarefa culmina com um **artefato parcial** corretamente implementado
  - termina-se o dia útil com um construto, mesmo que parcial, sem defeito conhecido
  - o esforço de diagnose e depuração tende a restringir-se a examinar somente o que foi adicionado na iteração

## DDT: desvantagens




- O custo da diagnose pode aumentar
  - Ao encontrar um problema tem-se muitos artefatos candidatos a conterem a(s) causa(s)
    - o módulo objetivo
    - o módulo de teste específico
    - a suíte de teste
    - os módulos já aprovados em iterações anteriores
    - o arcabouço (*framework*) de teste
- **Na prática o custo é bem menor**, pois o teste automatizado pode fornecer informação que facilita e agiliza a diagnose
  - desenvolvimento incremental tende a reduzir o volume de código a ser examinado ao diagnosticar uma falha
    - os possíveis defeitos tendem a estar no código adicionado ou alterado pelo incremento

Laboratório de Engenharia de Software

## DDT: desvantagens

- Mais coisa para co-evoluir → manter :
  - o módulo objetivo
  - o módulo de teste específico
  - as assertivas
  - os *scripts* de teste
  - raras vezes o *framework* de teste
    - múltiplas versões do *framework* podem criar problemas gerenciais
- Se a documentação for ruim isto pode tornar-se um problema complicado, conseqüentemente temos
  - mais coisa para documentar
  - mais documentação para verificar, validar e aprovar




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

Laboratório de Engenharia de Software

## DDT: desvantagens

- Nem sempre é possível utilizar teste automatizado, ex.
  - programas para os quais seja difícil criar oráculos automatizados, exemplos:
    - figuras, gráficos
    - leiaute de janelas
    - interfaces humano-computador
  - sistemas em que seja difícil de calcular o valor esperado
    - programas em que o algoritmo e a especificação são idênticos
      - ex. cálculo de juros variáveis compostos
    - sistemas que utilizam aprendizado
      - comportamento emergente
    - sistemas multi-programados / processados
      - agentes
    - sistemas distribuídos



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

## Behavior Driven Development



- DBC – desenvolvimento baseado em comportamento
  - BDD – *behavior driven development*
- O que é?
  - é uma técnica de desenvolvimento de software que evolui o *Test Driven Development*.
  - visa o teste funcional visando características (features)
  - busca desassociar TDD do conceito de testes de módulos
    - procura enfatizar o aspecto especificação subjacente ao TDD
  - busca aproximar participantes com pouco (sem?) conhecimento técnico ao processo de desenvolvimento
    - patrocinadores, clientes
    - usuários

## Behavior Driven Development



- Mas o que o teste tem a ver com a participação de clientes e/ou usuários?
  - uma das propostas de **processos ágeis** é que a especificação funcional seja produzida pelos usuários e/ou clientes
    - especificações devem ser verificáveis, ou seja **testáveis**
  - a proposta de DDT é que a **especificação funcional seja traduzida para módulos de teste**
    - o módulo de teste é uma especificação por meio de exemplos
      - usualmente visa módulos e não componentes, programas, ou sistemas
    - como é escrito em uma linguagem de programação, ou de script de teste, tende a ser difícil para usuários produzirem ou mesmo entenderem os módulos de teste
    - é comum DDT ser confundido com mecanismo de validação apenas

## Behavior Driven Development



- Como BDD procura resolver?
  - o usuário deve escrever a especificação
    - utiliza para isso uma linguagem natural (português) restrita
    - ver aula 12 Teste Funcional 3: Teste baseado em casos de uso
    - ver aula 13 Máquinas de estado: Teste baseado em máquinas de estado
  - espera que esta especificação seja o próprio teste
    - talvez precise de alguma complementação de código
  - mas isso força que o usuário final detenha conhecimentos de programação
    - observado na prática : não muita
  - para facilitar este processo, propõe-se linguagens que permitam o usuário escrever em uma linguagem quase natural
    - esta deve ser interpretada por uma ferramenta de testes
    - deve ser legível por pessoas com pouco conhecimento de programação

## Behavior Driven Development



- Algumas ferramentas conhecidas
  - JBehave (Java)
  - RSpec (Ruby)
  - Scriptactulous Unit Testing framework
  - Cucumber

## Behavior Driven Development



Empty Movie List (empty é verbo neste caso...)

```
list size should equal 0
list should not include "Star Wars"
```

- resulta em (Ruby)

```
require 'spec'
require 'movie'
require 'movie_list'
class EmptyMovieList < Spec::Context
  def setup
    @list = MovieList.new
  end
  def should_have_size_of_0
    @list.size.should_equal 0
  end
  def should_not_include_star_wars
    @list.should_not_include "Star Wars"
  end
end
```

Astels, D.; A New Look at Test-driven Development

Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

29

## Behavior Driven Development



One Movie in List

```
add movie "Star Wars"
list size should equal 1
list should include "Star Wars"
```

- Resulta em

```
class OneMovieList < Spec::Context
  def setup
    @list = MovieList.new
    star_wars = Movie.new "Star Wars"
    @list.add star_wars
  end
  def should_have_size_of_1
    @list.size.should_equal 1
  end
  def should_include_star_wars
    @list.should_include "Star Wars"
  end
end
```


Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

30

Laboratório de Engenharia de Software

## Problemas com BDD



- Focaliza excessivamente no fluxo principal
  - os testes tendem a negligenciar
    - erros de uso
    - condições de contorno
  - possível solução
    - uso de máquinas de estado ou de diagramas estado transição
  
- Risco de explosão combinatória
  - volume grande de seleções em uma característica
  - possível solução
    - definir cada característica através de uma máquina de estados com poucos estados possuindo interfaces precisamente definidas
    - cada estado pode ser implementado por outra máquina de estados em nível menor de abstração

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

Laboratório de Engenharia de Software

## Bibliografia



- Beck, K., *Extreme Programming Explained: Embrace Change*; Addison-Wesley; 1999
- Beck, K.; *Test-Driven Development by Example*; New York, NY: Addison-Wesley; 2003
- Caldeira, L.R.N.; *Geração Semi-automática de Massas de Testes Funcionais a Partir da Composição de Casos de Uso e Tabelas de Decisão*; Dissertação de Mestrado; PUC-Rio; 2010
- Fewster, M.; Graham, D.; *Software Test Automation*; Addison-Wesley; 1999
- Fowler, M.; *Refactoring: Improving the Design of Existing Code*; Addison-Wesley; 2000
- Hunt, A.; Thomas, D.; *Pragmatic Unit Test: in Java with JUnit*; Raleigh, North Carolina: The Pragmatic Bookshelf; 2004
- North, D.; *Introducing BDD*; 2006; Buscado em: 1/junho/2009; URL: <http://dannorth.net/introducing-bdd>
- Vlaskine, V.; *A Brief Overview of Software Methodologies*; 2003; Buscado em: 06/fevereiro/2008; URL: <http://www.geocities.com/softwarepeoplenet/resmeth.html>

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



Laboratório de Engenharia de Software

LES

FIM

Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

33