

Test Driven Development

Origem: Wikipédia, a enciclopédia livre.

Test Driven Development (TDD) ou em português **Desenvolvimento guiado por testes** é uma técnica de desenvolvimento de software que se relaciona com o conceito de verificação e validação e se baseia em um ciclo curto de repetições: Primeiramente o desenvolvedor escreve um caso de teste automatizado que define uma melhoria desejada ou uma nova funcionalidade. Então, é produzido código que possa ser validado pelo teste para posteriormente o código ser refatorado para um código sob padrões aceitáveis. Kent Beck, considerado o criador ou o 'descobridor' da técnica, declarou em 2003 que TDD encoraja designs de código simples e inspira confiança.^[1] Desenvolvimento dirigido por testes é relacionado a conceitos de programação de Extreme Programming, iniciado em 1999,^[2] mas recentemente tem-se criado maior interesse pela mesma em função de seus próprios ideais.^[3] Através de TDD, programadores podem aplicar o conceito de melhorar e depurar código legado desenvolvido a partir de técnicas antigas.^[4]

Índice

Requisitos

Ciclo de Desenvolvimento

- Adicione um teste
- Execute todos os testes e veja se algum deles falha
- Escrever código
- Execute os testes automatizados e veja-os executarem com sucesso
- Refatorar código
- Repita tudo

Estilos de Desenvolvimento

Benefícios

Limitações

Visibilidade de Código

Fakes, mocks e testes de integração

Ver também

Referencias

Notas

Ligações externas

Requisitos

Desenvolvimento dirigido por testes requer dos desenvolvedores criar testes de unidade automatizados que definam requisitos em código antes de escrever o código da aplicação. Os testes contém asserções que podem ser verdadeiras ou falsas. Após as mesmas serem consideradas verdadeiras após sua execução, os testes confirmam o comportamento correto, permitindo os desenvolvedores evoluir e refatorar o código. Desenvolvedores normalmente usam Frameworks de testes, como xUnit, para criar e executar automaticamente uma série de casos de teste.

Ciclo de Desenvolvimento

1. Adicione um teste

Em **Test Driven Development**, cada nova funcionalidade inicia com a criação de um teste. Este teste precisa inevitavelmente falhar porque ele é escrito antes da funcionalidade a ser implementada (se ele falha, então a funcionalidade ou melhoria 'proposta' é óbvia). Para escrever um teste, o desenvolvedor precisa claramente entender as especificações e requisitos da funcionalidade. O desenvolvedor pode fazer isso através de casos de uso ou user stories que cubram os requisitos e exceções condicionais. Esta é a diferenciação entre desenvolvimento dirigido a testes entre escrever testes de unidade 'depois' do código desenvolvido. Ele torna o desenvolvedor focado nos requisitos 'antes' do código, que é uma sutil mas importante diferença.

2. Execute todos os testes e veja se algum deles falha

Esse passo valida se todos os testes estão funcionando corretamente e se o novo teste não traz nenhum equívoco, sem requerer nenhum código novo. Pode-se considerar que este passo então testa o próprio teste: ele regula a possibilidade de novo teste passar.

O novo teste deve então falhar pela razão esperada: a funcionalidade não foi desenvolvida. Isto aumenta a confiança (por outro lado não exatamente a garante) que se está testando a coisa certa, e que o teste somente irá passar nos casos intencionados.

3. Escrever código

O próximo passo é escrever código que irá ocasionar ao teste passar. O novo código escrito até esse ponto poderá não ser perfeito e pode, por exemplo, passar no teste de uma forma não elegante. Isso é aceitável porque posteriormente ele será melhorado.

O importante é que o código escrito deve ser construído *somente* para passar no teste; nenhuma funcionalidade (muito menos não testada) deve ser predita ou permitida em qualquer ponto.

4. Execute os testes automatizados e veja-os executarem com sucesso

Se todos os testes passam agora, o programador pode ficar confiante de que o código possui todos os requisitos testados. Este é um bom ponto que inicia o passo final do ciclo TDD.

5. Refatorar código

Nesse ponto o código pode ser limpo como necessário. Ao re-executar os testes, o desenvolvedor pode confiar que a refatoração não é um processo danoso a qualquer funcionalidade existente. Um conceito relevante nesse momento é o de remoção de duplicação de código, considerado um importante aspecto ao design de um software. Nesse caso, entretanto, isso aplica remover qualquer duplicação entre código de teste e código de produção — por exemplo magic numbers or strings que nós repetimos nos testes e no código de produção, de forma que faça o teste passar no passo 3.

6. Repita tudo

Iniciando com outro teste, o ciclo é então repetido, empurrando a funcionalidade a frente. O tamanho dos passos deve ser pequeno - tão quanto de 1 a 10 edições de texto entre cada execução de testes. Se novo código não satisfaz rapidamente um novo teste, ou outros testes falham inesperadamente, o programador deve desfazer ou reverter as alterações ao invés do uso de excessiva depuração. A Integração contínua ajuda a prover pontos reversíveis. É importante lembrar que ao usar bibliotecas externas não é interessante gerar incrementos tão pequenos que possam efetivamente testar a biblioteca,^[3] a menos que haja alguma razão para acreditar que a biblioteca tenha defeitos ou não seja suficientemente completada com suas funcionalidades de forma a servir às necessidades do programa em que está sendo escrito.

Estilos de Desenvolvimento

Há vários aspectos ao usar desenvolvimento dirigido a testes, por exemplo os princípios "Keep It Simple, Stupid" (KISS) e "You Ain't Gonna Need It" (YAGNI). Focando em escrever código somente para os testes passarem, o design do sistema pode ser mais limpo e claro do que o que é alcançado por outros métodos.^[1] Em *Test-Driven Development By Example* Kent Beck sugere o princípio "Fake it, till you make it". Para alcançar altos níveis conceituais de design (como o uso de Design Pattern), testes são

escritos de forma que possam gerar o design. O código pode acabar permanecendo mais simples do que o padrão alvo, entretanto ele deve passar em todos os testes requeridos. Isto pode ser inaceitável de primeira mas ele permite o desenvolvedor focar somente no que é importante.

Falhar primeiro os casos de testes. A ideia é garantir que o teste realmente funciona e consegue capturar um erro. Assim que ele é mostrado, a funcionalidade almejada pode ser implementada. Isto tem sido apontado como o "Test-Driven Mantra", conhecido como vermelho/verde/refatorar onde vermelho significa *falhar* onde pelo menos uma asserção falha ,e verde é *passar*, que significa que todas as asserções foram verdadeiras.

Desenvolvimento dirigido a testes constantemente repete os passos de adicionar casos de teste que falham, passando e refatorando-os. Ao receber o resultado esperado em cada estágio reforça ao desenvolvedor o modelo mental do código, aumentando a confiança e incrementando a produtividade.

Práticas avançadas de desenvolvimento dirigido a testes encaminham para o desenvolvimento dirigido a testes de aceitação (ATDD), onde o critério especificado pelo cliente é automatizado em testes de aceitação, que então levam ao tradicional processo de desenvolvimento dirigido a testes de unidade. Este processo garante que o cliente tem um mecanismo automatizado para decidir como o software atende suas necessidades. Com ATDD, o desenvolvimento em equipe tem objetivo específico para satisfazer, e os testes de aceitação os mantém continuamente focados em o que o cliente realmente deseja daquela funcionalidade.

Benefícios

Um estudo de 2005 descobriu que usar TDD significava escrever mais testes, e logo, programadores que escreviam mais testes tendiam a ser mais produtivos.^[5] Hipóteses relacionando a qualidade de código e uma correlação direta entre TDD e produtividade foram inconclusivas.^[6]

Desenvolvedores usando TDD puramente em novos projetos reportaram que raramente necessitaram a utilização de um depurador. Usado em junção com um Sistema de controle de versão, quando testes falham inesperadamente, reverter o código para a última versão em que os testes passaram pode ser mais produtivo do que depurar.^{[7][8]}

Desenvolvimento dirigido por testes oferece mais do que somente um maneira simples de validação e de correção, pode orientar o design de um programa. Por focar em casos de testes primeiramente, deve-se imaginar como a funcionalidade será usada pelo cliente. Logo, o programador é somente com a interface e não com a implementação. Este benefício é complementar ao Design por Contrato, que através torna os casos de testes muito mais do que asserções matemáticas ou pré-concepções.

O poder que TDD oferece é a habilidade de pegar pequenas partes quando requeridas. Isso permite o desenvolvedor focar como objetivo fazer os testes atuais passarem. Casos excepcionais e tratamento de erros não são considerados inicialmente. Testes que criam estas circunstâncias estranhas são implementadas separadamente. Outra vantagem é que TDD quando usado apropriadamente, garante que todo o código desenvolvido seja coberto por um teste. Isto fornece a equipe de desenvolvimento, e ao usuários, subseqüentemente, um grande nível de confiança ao código.

Enquanto que é verdade que mais código é necessário ao usar TDD do que sem TDD, devido aos códigos de teste, o tempo total de implementação é tipicamente menor.^[9] Um grande número de testes ajudam a limitar o número de defeitos no código. A natureza periódica ajuda a capturar defeitos inicialmente no ciclo de desenvolvimento, prevenindo-os de se tornarem grandes e endêmicos problemas. Eliminando defeitos cedo no processo normalmente evita longos e tediantes períodos de depuração posteriores em um projeto.

TDD pode encaminhar a um nível que possibilite um código mais modularizado, flexível e extensível. Este efeito surge devido a metodologia requerer que os desenvolvedores pensem no software em pequenas unidades que podem ser re-escritas, desenvolvidas e testadas independentemente e integradas depois. Isto implica menores e mais classes, evitando o acoplamento e permitindo interfaces mais limpas. O uso de Mock Object é um contribuidor para a modularização do código, pois este recurso requer que o código seja escrito de forma que possa ser facilmente trocado entre versões Mock, usados em testes de unidade, e "reais", usados na aplicação.

Devido a fato de que nenhum código é escrito a não ser para passar em um teste que esteja falhando, testes automatizados tendem a cobrir cada caminho de código. Por exemplo, para que um desenvolvedor possa adicionar um caminho alternativo "senão" em um "se" , o desenvolvedor poderia primeiramente escrever um teste que motive o fluxo alternativo. Como resultado, os testes

automatizados TDD tendem a ser mais perfeitos: eles irão mostrar qualquer mudança inesperada no comportamento do código. Isto ajuda a identificar problemas cedo que poderiam aparecer ao consertar uma funcionalidade que ao modificada, inesperadamente altera outra funcionalidade.

Limitações

1. Desenvolvimento dirigido com testes é difícil de usar em situações onde testes totalmente funcionais são requeridos para determinar o sucesso ou falha. Exemplos disso são interfaces de usuários, programas que trabalham com base de dados, e muitos outros que dependem de configurações específicas de rede. TDD encoraja os desenvolvedores a incluir o mínimo de código funcional em módulos e maximizar a lógica, que é extraída em código de teste, usando Fakes mocks para representar o mundo externo.
2. Suporte gerencial é essencial. Se toda a organização não acreditar que TDD é para melhorar o produto, o gerenciamento irá considerar que o tempo gasto escrevendo teste é desperdício.^[10]
3. Os próprios testes se tornam parte da manutenção do projeto. Testes mal-escritos, por exemplo, que incluem strings de erro embutidas ou aqueles que são suceptíveis a falha, são caros de manter. Há um risco em testes que geram falsas falhas de tenderem a serem ignorados. Assim quando uma falha real ocorre, ela pode não ser detectada. É possível escrever testes de baixa e fácil manutenção, por exemplo pelo reuso das strings de erro, podendo ser o objetivo durante a fase de refatoração descrita acima.
4. O nível de cobertura e detalhamento de teste alcançado durante repetitivos ciclos de TDD não pode ser facilmente re-criado em uma data tardia. Com o passar do tempo os testes vão se tornando gradativamente preciosos. Se uma arquitetura pobre, um mal design ou uma estratégia de teste mal feita acarretar em mudança tardia, fazendo com que dezenas de testes falhem, por outro lado eles são individualmente consertáveis. Entretanto, simplesmente deletando, desabilitando ou alterando-os vagamente poderá criar buracos indetectáveis na cobertura de testes.
5. Lacunas inesperadas na cobertura de teste podem existir ou ocorrer por uma série de razões. Talvez um ou mais desenvolvedores em uma equipe não foram submetidos ao uso de TDD e não escrevem testes apropriadamente, talvez muitos conjuntos de testes foram invalidados, excluídos ou desabilitados acidentalmente ou com o intuito de melhorá-los posteriormente. Se isso acontece, a certeza é de que um enorme conjunto de testes TDD serão corrigidos tardiamente e refatorações serão mal acopladas. Alterações podem ser feitas não resultando em falhas, entretanto, na verdade, bugs estão sendo introduzidos imperceptivelmente, permanecendo indetectáveis.
6. Testes de unidade criados em um ambiente de desenvolvimento dirigido por testes são tipicamente criados pelo desenvolvedor que irá então escrever o código que está sendo testado. Os testes podem consequentemente compartilhar os 'pontos cegos' no código: Se por exemplo, um desenvolvedor não realizar determinadas entradas de parâmetros que precisam ser checadas, muito provavelmente nem o teste nem o código irá verificar essas entradas. Logo, se o desenvolvedor interpreta mal a especificação dos requisitos para o módulo que está sendo desenvolvido, tanto os testes como o código estarão errados.
7. O alto número de testes de unidades pode trazer um falso senso de segurança, resultando em menor nível de atividades de garantia de qualidade, como testes de integração e aceitação.

Visibilidade de Código

O conjunto de teste claramente possibilita acessar o código que está se testando. Por outro lado critérios normais de design como Information hiding, encapsulamento e separação de conceitos não devem ser comprometidos. Consequentemente teste de unidade são normalmente escritos no mesmo projeto ou módulo que o código está sendo testado.

Ao usar design orientado a objetos, este não provê acesso ao métodos e dados privados. Consequentemente, trabalho extra pode ser necessário para testes de unidade. Em Java e outras liguagens, um desenvolvedor pode usar reflexão para acessar campos que são marcados como privados.^[11] Alternativamente, uma classe inerente pode ser usada para suportar os testes de unidade, assim tendo visibilidade dos membros e atributos da classe. No .NET e em muitas outras linguagens, classes parciais podem ser usadas para expor métodos privados e dados para os testes acessarem.

O importante é que modificações brutas no testes não apareçam no código de produção. Em C# e em outras linguagens, diretivas de de pré-compilação como `#if DEBUG ... #endif` pode ser posicionadas em volta de classes adicionais e realmente prevenir todos os outros códigos relacionados a teste de serem compilados no código de lançamento. Isto então significa que o código lançado não é exatamente o mesmo de quando testado. A execução regular de poucos mais completos, testes de integração de aceitação ao final do lançamento do código podem garantir que não existe código de produção que subitamente dependa dos testes. Há muito debate entre os praticantes de TDD, documentados nos seus blogs e outros locais, com a dúvida se é de bom julgamento testar métodos privados e protegidos e dados de qualquer maneira. Muitos argumentam que poderia ser suficiente testar qualquer classe através da interface pública, considerando membros privados como meramente detalhes de implementações que podem mudar, e se deveria ser permitido fazer sem quebrar nenhum teste. Outros dizem que aspectos cruciais de uma funcionalidade

podem ser implementados em métodos privados, e que ao fazer desta forma eles são indiretamente testados através da interface pública obscurecendo a ocorrência dos mesmos: Testes de unidade são para testar a menor unidade de funcionalidade possível.^[12]^[13]

Fakes, mocks e testes de integração

Testes de unidades são assim chamados por cada teste exercitar uma *unidade* de código. Se um módulo tem centenas de testes de unidade ou somente cinco é irrelevante. Um conjunto de testes em TDD nunca cruza os limites de um programa, nem deixa conexões de rede perdidas. Ao fazer essas ações, o mesmo introduz intervalos de tempo que podem tornar testes lentos ao serem executados, desencorajando desenvolvedores de executar toda a suite de testes. Introduzir dependências de módulos externos ou data transforma *teste de unidade* em *testes de integração*. Se um módulo se comporta mal em uma cadeia de módulos interrelacionados, não fica imediatamente claro onde olhar a causa da falha.

Quando o código em desenvolvimento depende confiavelmente de uma base de dados, um serviço web ou qualquer outro processo externo ou serviço, obrigando em um separação unitária de teste é então uma oportunidade forçada de criar um desgin de código mais modular, mais testável e reusável.^[14] Dois passos são necessários:

1. Toda vez que um acesso externo é necessário no design final, uma interface deve ser definida de forma a descrever que acessos irão ser disponíveis.O princípio de inversão de dependência fornece benefícios nessa situação em TDD.
2. A interface deve ser implementada de duas maneiras, uma que realmente acessa o processo externo, e outra que é um fake ou mock. Objetos fake precisam fazer um pouco mais do que adicionar mensagens "Objeto Pessoa salvo" criando registro de rastreio, identificando que asserções podem executadas para verificar o comportamento correto. Mock objects diferem pelo fato que eles mesmos contém asserções que podem fazer com que o teste falhe. Exemplo:

Se o nome da pessoa e outro dado não é como esperado. métodos de objetos fake e mock que retornem dados, aparentemente de uma armazenamento de dados ou de usuário,pode ajudar ao processo de teste sempre retornar o mesmo, dados que testes pode confiar. Eles podem ser muito usáveis em predefinidos modos de falha em que rotinas de tratamento de erro pode ser desenvolvidas e confiavelmente testadas. Servicos fake dentre outros armazenamento de dados podem ser usáveis em TDD: Um serviço fake de criptografia pode não, criptografar o dado passado; serviços fake de número aleatório podem sempre retornar 1. Implementações de fakes e mocks são exemplos de injeção de dependência.

A proposta da injeção de dependência é que a base de dados ou qualquer ou código de acesso externo nunca é testado pelo processo TDD. Para desviar de erros que possam aparecer em função disso, outros testes com a real implementação das interfaces discutidas acima são necessários. Esses testes são separados dos testes unitários e são realmente testes de integração. Eles serão poucos, e eles precisam ser menos executados os testes de unidades. Eles podem ser implementados sem nenhum problema usando o mesmo framework de testes, como xUnit .


Testes de integração que alteram qualquer armazenamento persistente ou banco de dados deve ser designado cuidadosamente, levando em consideração os estados iniciais e finais dos arquivos e base de dados, independente se um teste falha. Isto é muitas vezes adquirido usando várias combinações das seguintes técnicas:

- O método TearDown, que é integrado ao muitos frameworks de teste.
- Estruturas try...catch...finally de tratamento de exceções, quando disponíveis.
- Transações de banco de dados onde ela atomicamente inclui talvez um escrita, uma leitura e uma operação de deleção.
- Capturando o "estado" do banco de dados antes de executar qualquer teste e após a execução, desfasar para o "estado" antes do teste ser executado. Isso pode ser automatizado usando a framework como o Ant ou NAnt ou um sistema de integração contínua como o CruiseControl.
- Inicializando o banco de dados para um estado limpo *antes* dos testes, ao invés de limpar *depois* da execução dos testes. Isto pode ser relevante onde limpar depois pode tornar difícl de diagnosticar falhas de teste ao deletar o estado final do banco de dados antes que um diagnóstico detalhado possa ser feito.

Framework como Moq, jMock, NMock, EasyMock, TypeMock, jMockit, Mockito, PowerMock e Rhino Mocks existem para tornar o processo de criar e usar objetos mock complexos facilmente.

Ver também

- Aegis um sistema de gerenciamento de mudanças de software que suporta um fluxo de trabalho de desenvolvimento de software dirigido por teste .



A Wikipédia tem o portal:

- [Behavior Driven Development](#)
- [Design por Contrato](#)
- [Lista de filosofias de desenvolvimento de software](#)
- [Lista de frameworks de teste unitário](#)
- [Mock object](#)
- [Teste de Software](#)
- [Caso de Teste](#)
- [Teste unitário](#)

Referencias

1. Beck, K. *Test-Driven Development by Example*, Addison Wesley - Vaseem, 2003
2. "Extreme Programming", *Computerworld* (online), December 2001, webpage: [Computerworld-appdev-92](http://www.computerworld.com/softwaretopics/software/appdev/story/0,10801,66192,00.html) (<http://www.computerworld.com/softwaretopics/software/appdev/story/0,10801,66192,00.html>).
3. Newkirk, JW and Vorontsov, AA. *Test-Driven Development in Microsoft .NET*, Microsoft Press, 2004.
4. Feathers, M. *Working Effectively with Legacy Code*, Prentice Hall, 2004
5. Erdogmus, Hakan; Morisio, Torchiano. «On the Effectiveness of Test-first Approach to Programming» (http://it-iti.nrc-cnrc.gc.ca/publications/nrc-47445_e.html). Proceedings of the IEEE Transactions on Software Engineering, 31(1). January 2005. (NRC 47445). Consultado em 14 de janeiro de 2008. "We found that test-first students on average wrote more tests and, in turn, students who wrote more tests tended to be more productive."
6. Proffitt, Jacob. «TDD Proven Effective! Or is it?» (<http://theruntime.com/blogs/jacob/archive/2008/01/22/tdd-proven-effective-or-is-it.aspx>). Consultado em 21 de fevereiro de 2008. "So TDD's relationship to quality is problematic at best. Its relationship to productivity is more interesting. I hope there's a follow-up study because the productivity numbers simply don't add up very well to me. There is an undeniable correlation between productivity and the number of tests, but that correlation is actually stronger in the non-TDD group (which had a single outlier compared to roughly half of the TDD group being outside the 95% band)."
7. Clark, Mike. «Test-Driven Development with JUnit Workshop» (<http://clarkware.com/courses/TDDWithJUnit.html>). Clarkware Consulting, Inc. Consultado em 1 de novembro de 2007. "In fact, test-driven development actually helps you meet your deadlines by eliminating debugging time, minimizing design speculation and re-work, and reducing the cost and fear of changing working code."
8. Llopis, Noel (20 de fevereiro de 2005). «Stepping Through the Looking Glass: Test-Driven Game Development (Part 1)» (<http://www.gamesfromwithin.com/articles/0502/000073.html>). Games from Within. Consultado em 1 de novembro de 2007. "Comparing [TDD] to the non-test-driven development approach, you're replacing all the mental checking and debugger stepping with code that verifies that your program does exactly what you intended it to do."
9. Müller, Matthias M.; Padberg, Frank. «About the Return on Investment of Test-Driven Development» (<http://www.ipd.uka.de/mitarbeiter/muellerm/publications/edser03.pdf>) (PDF). Universität Karlsruhe, Germany. 6 páginas. Consultado em 1 de novembro de 2007
10. Loughran, Steve (November 6th, 2006). «Testing» (<http://people.apache.org/~stevel/slides/testing.pdf>) (PDF). HP Laboratories. Consultado em 12 de agosto de 2009 Verifique data em: |data= (ajuda)
11. Burton, Ross (11 de dezembro de 2003). «Subverting Java Access Protection for Unit Testing» (<http://www.onjava.com/pub/a/onjava/2003/11/12/reflection.html>). O'Reilly Media, Inc. Consultado em 12 de agosto de 2009
12. Newkirk, James (7 de junho de 2004). «Testing Private Methods/Member Variables - Should you or shouldn't you» (<http://blogs.msdn.com/jamesnewkirk/archive/2004/06/07/150361.aspx>). Microsoft Corporation. Consultado em 12 de agosto de 2009
13. Stall, Tim (1 de março de 2005). «How to Test Private and Protected methods in .NET» (<http://www.codeproject.com/KB/cs/testnonpublicmembers.aspx>). CodeProject. Consultado em 12 de agosto de 2009
14. Fowler, Martin (1999). *Refactoring - Improving the design of existing code*. Boston: Addison Wesley Longman, Inc. ISBN 0-201-48567-2

Notas

- *Este artigo foi inicialmente traduzido do artigo da Wikipédia em inglês, cujo título é «[Test Driven Development](#)».*

Ligações externas

- [c2.com](http://c2.com/cgi/wiki/TestDrivenDevelopment) (<http://c2.com/cgi/wiki/TestDrivenDevelopment>) Test-driven development from WikiWikiWeb
- Test or spec? Test and spec? Test from spec! (http://www.eiffel.com/general/monthly_column/2004/september.html), by Bertrand Meyer (September 2004)
- Microsoft Visual Studio Team Test from a TDD approach ([http://msdn.microsoft.com/en-us/library/ms379625\(VS.80\).aspx](http://msdn.microsoft.com/en-us/library/ms379625(VS.80).aspx))
- Write Maintainable Unit Tests That Will Save You Time And Tears (<http://msdn.microsoft.com/en-us/magazine/cc163665.aspx>)
- TDD generico que funciona em qualquer ferramenta existente (<https://sites.google.com/site/rodrigonw/home/reuseofcomponentsforanytoolautomationandperformance>)

Esta página foi editada pela última vez às 18h16min de 3 de janeiro de 2019.

Este texto é disponibilizado nos termos da licença Atribuição-Compartilhagual 3.0 Não Adaptada (CC BY-SA 3.0) da Creative Commons; pode estar sujeito a condições adicionais. Para mais detalhes, consulte as condições de utilização.