6
Test Driven Development

O processo de teste de um software é uma maneira de verificar se este está correto, completo e qual o seu nível de qualidade. Por isso, este é um procedimento indispensável no desenvolvimento de qualquer software.

Mas, tradicionalmente, como os testes são feitos? O estilo de desenvolvimento de software em cascata (do inglês *waterfall*) Royce (1970) propõe uma fase de verificação bem próxima do término do projeto. A Figura 8 apresenta as etapas do modelo de desenvolvimento em cascata. "Isto é curioso, pois se sabe que nesta etapa do desenvolvimento de um software, o custo de uma mudança no código ou nos requisitos é comprovadamente maior" (Murphy, 2005). "Muitas companhias descobriram que focar na correção dos defeitos de um projeto mais cedo pode reduzir os custos de desenvolvimento e o cronograma pela metade ou até mais" (McConnell, 2004).

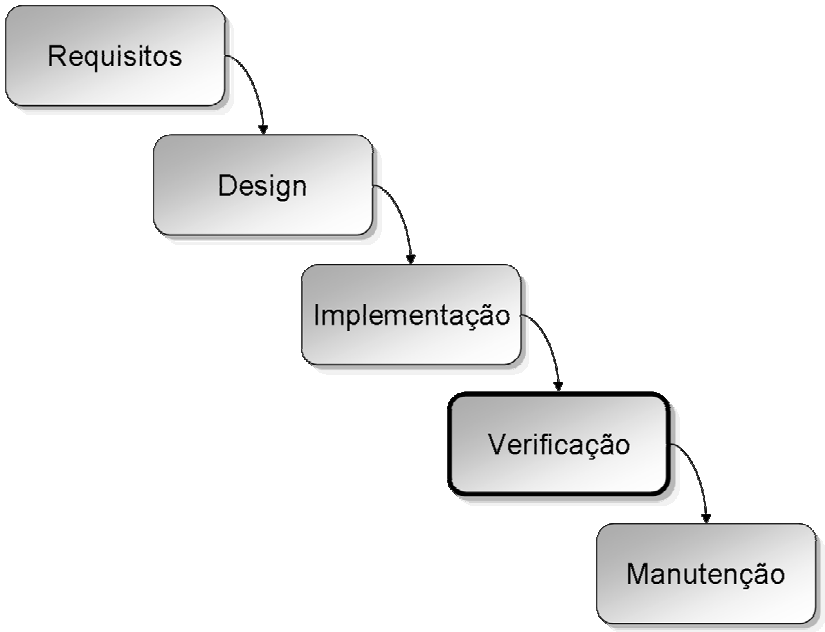

Figura 8: Modelo em cascata. Verificação é a penúltima etapa.

A Figura 9, apresentada inicialmente por Barry Boehm[24], mostra como o custo para corrigir um defeito fica cada vez mais caro com o decorrer do projeto. Boehm propôs, então, o modelo em espiral para amenizar este tipo de problema. Este modelo propõe uma forma de desenvolvimento iterativa e incremental. Os desenvolvedores passam várias vezes pelas fases de análise de requisitos, design, implementação, verificação e distribuição. Dessa forma, é possível minimizar o custo de mudanças e manutenção.

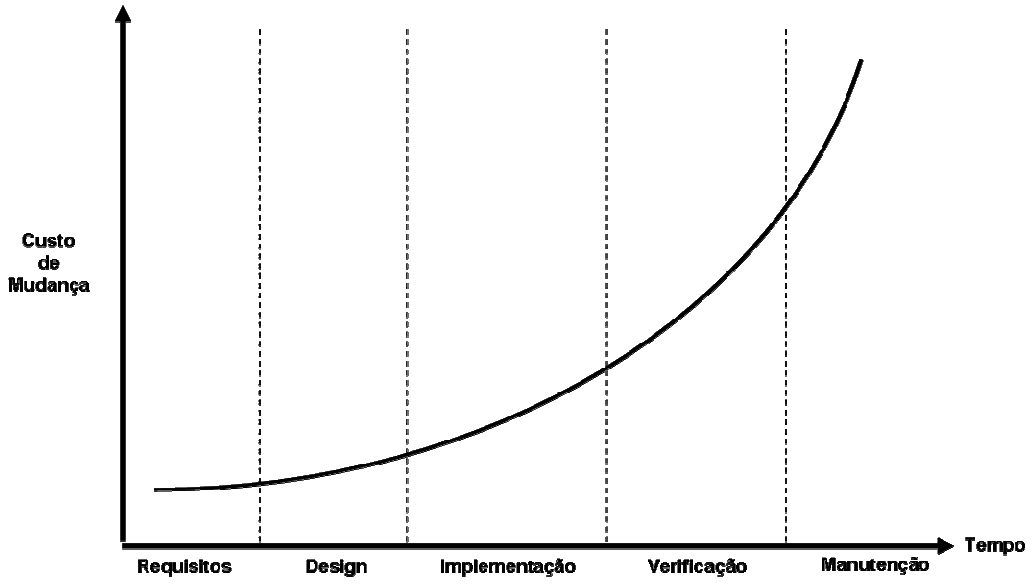

Figura 9: Custo de mudança usando abordagem tradicional.

Todos os envolvidos no desenvolvimento de um software – dos desenvolvedores aos clientes – concordam: testar é uma boa prática (Astels, 2003). Então, por que tantos sistemas são tão mal testados? Existem muitos problemas com a abordagem tradicional de teste:

- ***Os testes são feitos depois que todo o código foi escrito***. Quando alguém não está acostumado com um sistema, é preciso algum tempo e esforço para poder tratar dos problemas existentes no código. Mesmo se o próprio programador que fez o código tiver que testá-lo, caso ele faça isso depois de ter implementado há muito tempo, ele terá dificuldades.
- ***Os testes são feitos por outros desenvolvedores***. Muitas vezes, quando os testes são feitos depois da implementação, é uma equipe especializada que os faz. Como eles podem não entender algum detalhe do código, é possível que esqueçam testes importantes.

[24] http://c2.com/cgi/wiki?ExponentialCostCurve

- ***Os testes podem não se basear no código***. Os responsáveis por escrever os testes podem se basear em uma documentação ou em algum outro artefato que não seja o código. Se algum desses artefatos estiver desatualizado, os testes podem ser escritos de maneira totalmente equivocada.
- ***Os testes podem não ser automatizados***. Se os testes forem executados manualmente, certamente não serão realizados freqüentemente e exatamente da mesma maneira a cada vez.
- ***Alterações podem criar problemas que não são detectados pelos testes***. É perfeitamente possível consertar um problema usando uma abordagem tradicional de modo a criar falhas em outros lugares. Se a cobertura de testes do código não for completa (ou próxima disso), a infra-estrutura de testes existente pode não encontrar esses novos problemas.

Test Driven Development (TDD) resolve esses problemas e alguns outros. Também conhecida como programação ou desenvolvimento em que se escreve um teste primeiro, esta é uma abordagem incremental que envolve a criação de um caso de teste anteriormente à implementação do código necessário para que este passe. Depois disso, o teste é executado para provar que este realmente falha. Usando o conceito de design simples (do inglês *simple design*[25]) o método é implementado de forma a fazer com que o teste de unidade passe. Uma vez que isso aconteça, o programador usa o *refactoring* para limpar o código e mantê-lo sob as regras de design simples. Esses passos são feitos sucessivamente, até que cada funcionalidade esteja pronta.

Então, escrever testes primeiro é uma forma de testar um sistema? Segundo Ward Cunningham, na verdade, escrever testes primeiro não é uma técnica de

[25] *Simple design* é um princípio de XP e quer dizer que o design de um método, classe ou sistema deve ser o mais simples possível. Por exemplo, se um cliente concordar que na primeira versão apenas o usuário "teste" com a senha "123" vai ter acesso a todo o sistema, somente o código exato para que esta funcionalidade seja implementada deve ser escrito. Preocupações com sistemas de autenticação e restrições de acesso não importam neste momento. Um erro comum, no entanto, por parte dos programadores é confundir código simples com código fácil de escrever. Código fácil de escrever é todo aquele que é escrito sem preocupações de design. Nem sempre este tipo de código levará à solução mais simples de design. Esse entendimento é fundamental para o bom andamento de XP. Por isso, todo código fácil de escrever deve ser identificado e substituído através de *refactoring* por código simples.

teste (Beck, 2001a). Apesar de não parecer óbvio por causa do nome, o objetivo real do TDD é especificação e não validação (Ambler, 2003b). Em outras palavras, é uma maneira de pensar no design antes de escrever o código funcional. Ron Jeffries oferece uma visão complementar, dizendo que o objetivo do TDD é simplesmente escrever código limpo e que funcione (Ambler, 2003b).

Portanto, escrever testes primeiro é, na verdade, uma técnica de análise (Beck, 2001a). O desenvolvedor decide o que vai programar e o que não vai programar. Ademais, ele define as respostas que espera para cada situação. Escolher o que está no escopo e, mais do que isso, o que está fora do escopo, é crítico para o desenvolvimento de software. Escrever testes primeiro força o desenvolvedor a determinar explicitamente quais as circunstâncias foram levadas em consideração enquanto ele escrevia o código.

Além disso, escrever testes primeiro também é uma maneira de pensar no design lógico (Beck, 2001a). Quando se começa, não existe implementação. O código que é escrito nos casos de testes é apenas uma manifestação exterior de uma lógica que ainda não existe e que está em vias de ser criada. O desenvolvedor tem que pensar em como os vários objetos serão usados ao invés de pensar numa maneira mais fácil de implementar cada classe. O efeito colateral, tornar a classe mais testável, é um bônus do ponto de vista da programação (McBreen, 2002).

Este tipo de metodologia não é uma idéia nova. Há muito tempo programadores especificam as entradas e as possíveis saídas que um programa deve gerar antes de dar início à programação propriamente dita. *Test Driven Development* é um apanhado dessa antiga idéia combinado com novas linguagens e ferramentas de programação para proporcionar o desenvolvimento de código funcional e de boa qualidade.

Escrever testes primeiro trata das seguintes características:

- ***Foco e Escopo***: Determinando explícita e objetivamente o que um programa deve fazer, torna-se mais fácil focar na tarefa de codificação. O escopo fica controlado. Se for necessário adicionar um código para o caso de situações específicas, é só escrever um novo teste e depois fazê-lo passar.
- ***Acoplamento e coesão***: Se for muito difícil escrever um teste, é sinal de que existe um problema de design, não um problema relacionado com testes. Código coeso e pouco acoplado tende a ser mais fácil de ser testado.

- ***Feedback***: O desenvolvimento sucessivo de testes reduz o tempo para corrigir erros porque diminui o tempo para descobri-los. A utilização do desenvolvimento dirigido por testes continuamente permite que os testes sejam executados a cada mudança no programa, assim como um compilador é executado a cada mudança no código-fonte. As falhas encontradas nos testes são reportadas rapidamente, da mesma forma que os erros de compilação.
- ***Ritmo***: É fácil ficar perdido por horas quando se está programando. A programação baseada em testes torna claro o que deve ser feito a seguir: escrever outro teste ou fazer passar um teste que está falhando. Com o tempo, isso se torna um ritmo natural de testar, codificar, fazer *refactoring*, testar, codificar, fazer *refactoring* e assim por diante.
- ***Confiança***: É difícil acreditar no autor de um código que não funciona. Escrever código limpo, que funciona e demonstrar as intenções por meio de testes automatizados aumenta a confiança entre os programadores de uma equipe.
- ***Cobertura de Testes***: Se um *bug* é introduzido durante uma alteração, um teste que cubra todo o código provavelmente irá encontrá-lo e detalhar a sua localização. Com o tempo, a tendência é que a cobertura se torne cada vez mais completa. E isso não é uma preocupação com os testes, é apenas uma conseqüência de se usar a técnica.

"Os testes também oferecem uma medida de progresso" (Beck & Andres, 2004). Quando um desenvolvedor escreve o teste, ele já progrediu uma vez que precisou pensar no design da funcionalidade que está sendo testada e teve que implementar este design. Além disso, ele já obteve um teste para uma funcionalidade. Quando este teste falha na primeira vez, o desenvolvedor também progrediu, pois ele sabe que ainda não concluiu o seu trabalho e possui um indício do que ainda precisa ser feito (o motivo de o teste ter falhado). "Se ele tem dez testes falhando e conserta um, então ele também fez progresso. Mais importante, ele tem uma medida clara de sucesso quando um teste não falha mais" (Ambler, 2003a).

Isto posto, recomenda-se que exista apenas um teste falhando por vez. Quando programar com testes primeiro, deve-se escrever um teste que falha, fazê-

lo funcionar e só depois passar para o próximo teste que falha. Isso permite ao desenvolvedor avançar de maneira gradual. Dessa forma, TDD aumenta a confiança de que o sistema está realmente atendendo aos requisitos definidos, que este realmente funciona e que se pode prosseguir com segurança.

Periodicamente o desenvolvedor vai executar todos os testes para ter certeza de que tudo continua funcionando conforme o esperado. Ao executar o aglomerado inteiro de testes de unidade, o programador está fazendo um teste de regressão completo do sistema. Se uma alteração produzir indiretamente alguma falha, os testes vão apontar o problema permitindo que este seja identificado e corrigido. Este tipo de verificação permite ao desenvolvedor tentar diversas soluções. Se os testes acusarem falhas, isso pode significar que a sua abordagem está incorreta e uma implementação diferente é necessária.

É mais demorado desenvolver usando esta prática? Essa é uma pergunta bastante pertinente. “Sem muito julgamento, levando-se em consideração apenas a tarefa de digitação do código, pode-se dizer seguramente que escrever casos de teste antes do código leva o mesmo tempo e esforço que escrever casos de teste logo após o código” (McConnell, 2004). A vantagem de escrever os testes primeiro é que isso encurta o ciclo de detecção de defeitos, depuração e correção. Logo, partindo do pressuposto acima, escrever os testes e o código certamente demora mais do que escrever apenas o código, já que existe mais código para ser escrito. A diferença é que este “tempo perdido” é recompensado quando há a necessidade de correção de um erro ou de um *refactoring*. Além disso, com esta prática, uma infra-estrutura de testes estará pronta no final do projeto. Isso não acontece quando se escreve apenas o código. Resumidamente, as vantagens de escrever os testes primeiro são:

- Diminui o tempo entre a inserção de um defeito no código, a sua detecção e, posteriormente, sua correção.
- Não aumenta o esforço em comparação com escrever casos de teste logo em seguida do código. É apenas uma seqüência diferente para a mesma atividade.
- Os defeitos são percebidos mais rapidamente e podem ser corrigidos com mais facilidade.

- Reduz muito o tempo gasto com depuração durante e depois de o código estar implementado.
- Tende a produzir código de melhor qualidade, pois força o programador a pensar pelo menos um pouco sobre os requisitos e o design antes de escrever o código.
- Expõe problemas nos requisitos funcionais mais cedo, antes de o código ser escrito, pois é muito difícil escrever um caso de testes quando os requisitos não estão bem definidos.
- Além de serem executados antes, os casos de teste também podem ser executados depois que o código estiver pronto.

De maneira geral, o desenvolvimento dirigido por testes é uma das práticas contemporâneas mais benéficas para produzir código de forma mais fácil e com mais qualidade.

6.1.Ciclo de Desenvolvimento

Qual é o primeiro passo para começar a testar um sistema? É preciso definir o que tem que ser feito. Por isso, antes de começar, um *brainstorm*[26] deve ser feito para definir uma listagem dos testes que precisarão ser criados. Além de descrever os requisitos de forma precisa, uma lista com os testes também indica o escopo da atividade, ajuda a manter o foco e serve como um critério para determinar se uma tarefa foi concluída (Newkirk & Vorontsov, 2004). Se um caso de teste que não foi imaginado anteriormente passar a existir, este deve ser adicionado à lista.

Uma sugestão é escrever a lista de testes sob a forma de expressões TODO como comentário do caso de teste que tem que ser escrito. A maioria das IDEs tratam estas expressões de forma especial, criando uma lista de tarefas que ainda precisam ser feitas. Dessa forma, qualquer pessoa que obtenha o código saberá claramente quais são as tarefas que ainda não foram realizadas.

---

[26] A tradução literal seria "tempestade cerebral". *Brainstorm* é uma técnica de reunião coletiva que consiste em agrupar pessoas de diferentes especialidades para uma discussão livre e descontraída. Os participantes podem expor qualquer idéia, por mais absurda que pareça, sobre todos os aspectos relacionados a um projeto. Neste caso, o *brainstorm* serve para discutir que funcionalidades serão implementadas e quais as expectativas com relação ao que o sistema deve fazer.

Se estiver muito complicado elaborar uma lista com os testes, isso significa que existe um problema com a definição dos requisitos. Se os requisitos não estiverem claros, fica impossível pensar nos testes. Essa é uma grande vantagem do TDD. Se não está evidente o que deve ser testado, o programador nem começa o seu trabalho. Ele terá que voltar para a etapa de entender verdadeiramente o problema, ao invés de ficar tentando uma solução para um problema impreciso.

Após definir a lista com os testes, deve-se escolher um teste e seguir o ciclo de desenvolvimento proposto pelo TDD, que é composto por cinco passos básicos (Beck, 2002):

1. ***Adicionar um teste rapidamente***

   O primeiro passo sempre será escrever um caso de teste. Para escrever o caso de teste, o desenvolvedor deve primeiro entender o que deve ser produzido e como ele vai chegar a esse resultado. Ele terá que escrever como espera usar as funcionalidades que vai testar neste exato momento.

2. ***Executar o teste para que este falhe***

   O caso de teste deve ser executado assim que for definido, mesmo que não exista implementação para que este passe. Este passo serve como uma calibragem para o caso de teste. Se o novo caso de teste passar, sem qualquer modificação no código do programa, então o teste pode estar errado ou ser desnecessário.

3. ***Alterar o mínimo de código para que o teste passe***

   Após verificar que existe um teste que falha, é necessário escrever o código para que este passe. Neste ponto, deve ser escrito o mínimo de código para o teste ser executado com sucesso. Esta característica do TDD faz com que o desenvolvimento seja feito através de pequenos incrementos, formados por poucos métodos e pouca lógica. Ao final, esses incrementos juntos produzem uma solução completa.

4. ***Executar o teste e ver a barra verde***

   O próximo passo é executar os casos de teste automatizados e observar se eles passam ou não. Em caso de sucesso, o programador terá certeza de que o código implementado por ele atende aos requisitos testados. Caso contrário, o código ainda precisa ser modificado.

5. *Eliminar o código duplicado*

O último passo é o *refactoring* e a limpeza do código duplicado. Nesta etapa nenhuma modificação relacionada à lógica do sistema deve ser feita. Apenas modificações estruturais são permitidas. Depois disso, os casos de teste devem ser executados mais uma vez para garantir que o *refactoring* não danificou alguma parte do sistema que já estava funcionando anteriormente.

Estes são os passos do ciclo de desenvolvimento dirigido por testes. Eles parecem simples – e com certeza são – mas quando se inicia o uso desta abordagem fica evidente a necessidade de muita disciplina. Distrair-se e escrever código funcional sem ter escrito um novo teste antes é muito fácil. "Uma das vantagens de usar *Pair Programming* no aprendizado de TDD é que os pares se vigiam para se manter na linha" (Amber, 2003a).

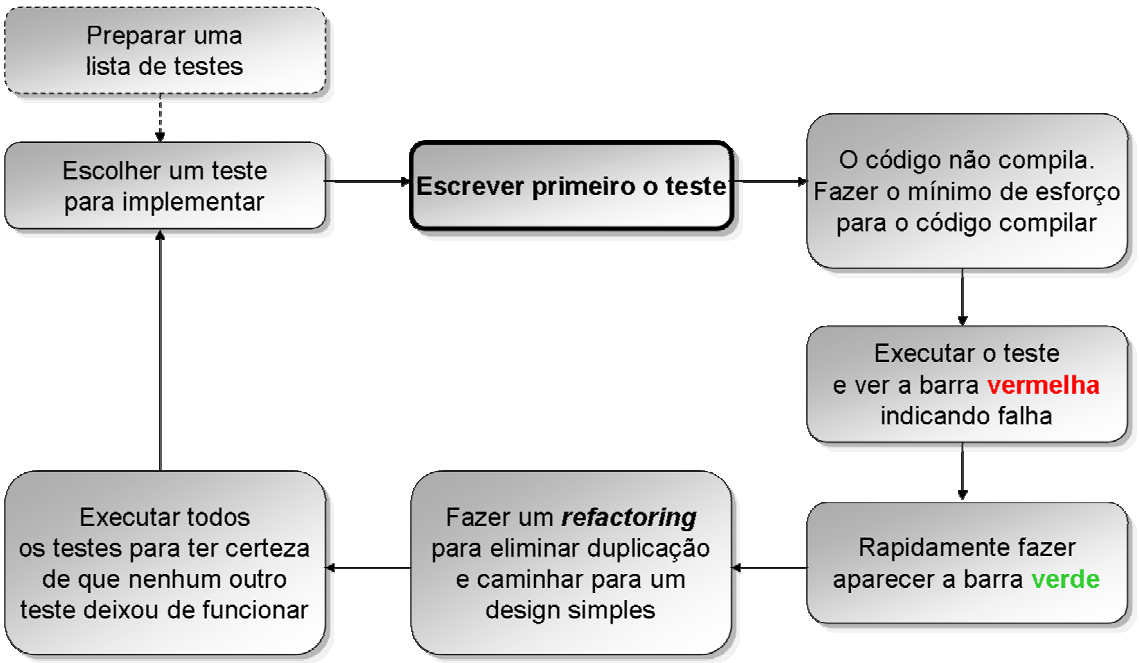

Figura 10: Ciclo de desenvolvimento dirigido por testes.

Além desses passos básicos, Beck (2002) determina que duas regras simples devem ser seguidas: 1) ***NUNCA escrever código se não houver um teste automatizado que falhe*** e 2) ***SEMPRE remover as duplicações***.

Alguns dos conceitos relacionados com o ciclo de desenvolvimento dirigido por testes, como barra vermelha e barra verde, estão intimamente ligados aos executores gráficos dos testes de unidade baseados em *frameworks* xUnit. Estes exibem uma barra verde quando todos os testes passam e uma barra vermelha

quando algum teste falha. Isso não significa que todas as ferramentas e os *frameworks* para testes vão produzir o mesmo efeito.

"Dependência é o problema chave em desenvolvimento de software, em todas as escalas" (Beck, 2002). Se dependência é o problema, duplicação é o sintoma. Duplicação normalmente tem a forma de lógica repetida – a mesma expressão aparecendo em vários lugares do código. A orientação a objetos é excelente para abstrair a duplicação da lógica e, neste caso, eliminar os sintomas acaba com os problemas. Por isso, abolir a duplicação em programas acaba com a dependência. É por isso que a segunda regra aparece em TDD. Eliminar a duplicação antes de passar para o próximo teste aumenta a chance de ter o próximo teste funcionando com uma única alteração (Beck, 2002).

Mas por qual teste começar? A recomendação é iniciar os testes pela variante da operação que não faz nada (Beck, 2002). Ou seja, se o teste for feito para uma lista, deve-se começar com uma lista vazia. Se for uma rotina recursiva, deve-se começar pelo caso base. Se o problema envolve uma iteração por objetos, deve-se começar testando para um único objeto.

Por onde começar a escrever um teste? Os programadores devem começar a escrever os testes pelos resultados esperados. Isso é feito utilizando-se métodos especiais da *framework* de testes para verificação do resultado (no caso do JUnit, os métodos com prefixo `assert`). São estas verificações que irão passar quando o teste estiver pronto e a funcionalidade implementada (Beck, 2002). As assertivas vão dizer o que realmente se espera que seja feito. Elas vão garantir que se algo inesperado for obtido, o teste falhará.

O que deve ser testado? Teste tudo aquilo que possa quebrar (Jeffries et al, 2001). Isso não significa que cada pedaço do código tem que ser testado minuciosamente. Na maior parte dos casos, testar os métodos *getters* e *setters*, por exemplo, é desnecessário. Teste toda a lógica que pode dar problemas, as condições excepcionais e as situações limite. Com a prática, o bom senso se torna a melhor forma de determinar o que deve ser testado e quais são os testes mais significativos.

O que fazer quando um defeito é reportado? Apesar de o uso de TDD proporcionar uma cobertura de testes de quase 100% do código, erros podem aparecer. Neste caso, "é fundamental escrever um pequeno teste que falhe

demonstrando o problema que foi reportado. Feito isso, é só repará-lo" (Beck, 2002).

O que fazer quando se sentir perdido? Beck (2002) sugere que tudo seja jogado fora e refeito. Isso pode ser um tanto drástico. Normalmente, se o caso de teste está bem especificado, é a falta de conhecimento do programador sobre o assunto que atrapalha. Às vezes ele não possui o domínio sobre a API que está sendo utilizada ou não sabe como funcionam as bibliotecas necessárias para resolver o problema ou simplesmente não tem idéia de como estas podem trabalhar em conjunto. Portanto, um caminho mais equilibrado é parar com os testes, criar um projeto simples (um Olá Mundo[27]) e fazer experimentos sem se preocupar com os testes. O importante é chegar a uma solução, mesmo que seja de uma forma detestável e inacabada. Assim, o programador ganha mais perspicácia para resolver o problema. Este novo projeto pode ser usado como consulta para implementar a solução real e depois deve ser jogado fora.

Finalmente, quando a lista de funcionalidades estiver vazia é um bom momento para revisar o design (Beck, 2002). Os conceitos foram representados de forma correta? Existe alguma duplicação que seja difícil de eliminar por causa do design atual? Manter o design e o código sempre no melhor estado e o mais simples possível é a chave para a evolução ser feita de maneira tranqüila. Isso pode ser chamado de "manutenção preventiva", uma vez que contribui para a eliminação de faltas latentes não observadas, e facilita futuras evoluções.

Seguindo esses procedimentos simples, o design do sistema vai sendo formado naturalmente e o código executável provê feedback instantâneo sobre as decisões tomadas. O programador escreve os seus próprios testes, já que não é possível esperar a cada iteração do ciclo de desenvolvimento por alguma outra pessoa que o faça. O design obtido tende a ser altamente coeso e com componentes pouco acoplados, o que faz com que testar não seja uma tarefa árdua, além de tornar a evolução e a manutenção do sistema mais fáceis.

---

[27] O Olá Mundo (do inglês *Hello World*) é um famoso programa de teste inicial de uma linguagem de programação. É um programa de computador que imprime a mensagem "*Hello, world*!" (Olá Mundo!) no dispositivo de saída. É utilizado em muitos manuais de introdução às linguagens de programação e com ele os estudantes costumam ter suas primeiras experiências de aprendizado. Também pode ser utilizado para definir um programa muito simples, que executa um exemplo básico de uso de uma API, por exemplo.

6.2.Qualidades de um Bom Caso de Teste

Apenas escrever casos de teste não é suficiente para garantir que um sistema esteja funcionando corretamente. Além disso, não é qualquer tipo de teste que viabiliza a utilização do TDD. Testes mal feitos, além de não revelarem possíveis problemas, podem encobrir faltas. Um bom caso de teste deve possuir algumas características que garantam sua efetividade e a agilidade na sua utilização. Abaixo estão listadas algumas dessas características:

Decisivos: Um teste deve conter toda informação necessária para determinar automaticamente se houve sucesso ou falha. Ou seja, não deve haver necessidade de uma inspeção visual para determinar o status do teste. O teste também deve ser expresso de forma que produza uma resposta simples (passou/falhou) ao invés de um resultado quantitativo ou qualitativo. Testes decisivos são expressos normalmente por meio de assertivas.

Válidos: O resultado de um teste deve ser verdadeiro. Isso quer dizer que, se um teste falhar, isso significa que existe um erro no artefato que está sendo testado. Se o teste passar, então não existe falha alguma no artefato testado.

Completos: Um teste contém todas as informações necessárias para ser executado corretamente e não requer nenhuma entrada externa para que rode. Todas as atividades relacionadas com o teste são executadas pelo próprio teste. Isso não significa que um teste não possa receber informações por parâmetro ou se basear em um arquivo para realizar os testes. Além disso, um teste deve ser capaz de fazer as modificações necessárias para que qualquer dependência que tenha sido alterada volte ao seu estado original, sem intervenção humana.

Reproduzíveis: Um teste sempre deve fornecer o mesmo resultado se o corpo do teste e o artefato que estiver sendo testado forem os mesmos. O teste é criado de forma que o resultado seja determinístico. Ainda que o sistema testado não o seja, um teste deve ser criado levando-se isso em conta e produzindo um resultado exato.

Isolados: O resultado de um teste não pode ser afetado pela execução de outros testes que sejam executados anteriormente. Além disso, um teste não pode afetar o resultado de testes que se sucedem. Logo, um conjunto de testes pode ser executado em qualquer ordem e o resultado será sempre o mesmo. Se, de alguma

forma, um teste depender do resultado ou dos efeitos de outros testes, então este não é isolado.

Automatizados: Um teste precisa de um único sinal para que seja executado por completo em um período finito de tempo. Não é necessária qualquer intervenção manual após o teste ter iniciado. Testes automatizados podem ser colocados juntos em *suites* de teste e podem ser executados sem interferência manual, um após o outro. A automação de testes depende de um sistema para controle de testes.

Rápidos: A execução de um teste não pode ser demorada. Não é em um teste de unidade o lugar mais adequado para avaliar a carga ou a performance de um sistema. O conjunto de testes deve ser executado em poucos segundos. Se um teste demorar mais do que isso, como ele poderá ser executado a cada pequena alteração que é feita no código?

Os desenvolvedores que conseguem criar testes com essas características estão mais próximos de fazer testes realmente efetivos. Isso porque estes vão rodar rapidamente, de forma isolada, usando informações que tornam tanto a leitura e quanto o entendimento mais fáceis e representando um passo em direção ao objetivo final.

6.3.Critérios para Seleção de Casos de Teste

Testes não podem garantir a ausência de erros. Isso significa que um programa nunca pode ser completamente testado. Por isso, é importante descobrir que subconjunto de todos os casos de teste tem a maior probabilidade de detectar a maior parte dos erros. Escolher ao acaso as entradas que serão utilizadas para testar uma funcionalidade, na maioria dos casos, não levará a esse resultado. Uma solução, então, é pensar bem nos casos de teste que serão escritos e utilizar critérios para selecioná-los:

- ***Saber qual o resultado esperado***. Antes de começar a escrever qualquer teste, o programador precisa saber o que deve acontecer após a execução do código que faz o teste passar. Se essa pergunta não puder ser respondida satisfatoriamente, então escrever o teste ou o código será perda de tempo. Caso isso aconteça e o programador não possa imaginar uma solução, então é hora de parar e verificar o que realmente deve ser feito. Por isso, o

primeiro critério para escrever um caso de teste é não escrevê-lo se os requisitos estiverem mal especificados.

- ***Verificar as condições limite***. A experiência mostra que casos de teste que exploram estas condições são muito mais proveitosos do que aqueles que não o fazem. Condições limite são aquelas situações anteriores, posteriores ou exatamente no limite de uma possível entrada. "Por exemplo, ao testar uma função matemática, os números -1, 0, 1 e o maior inteiro aceitável são entradas mais valiosas do que 5, 20, 79 ou 210. O mesmo ocorre com Strings. É mais interessante testar utilizando uma String vazia e, depois, com um conjunto muito grande de caracteres" (Link & Frolich, 2003). Condições limite não se restringem aos parâmetros de entrada. Estas podem ser também o tamanho de um arquivo, a quantidade máxima de conexões ou o número de objetos em uma coleção, por exemplo.
- ***Testar os possíveis caminhos***. Se um código possui condicionais, os testes devem ser escritos de forma a satisfazer cada uma das condições. Com o uso de TDD, é esperado que todo o código esteja coberto pelos testes, mas, se isso não acontecer, deve-se escrever um teste para considerar uma condição que não está sendo testada. "Este critério, no entanto, não poderá ser atendido sempre" (Myers, 2004).
- ***Testar os casos de erro***. No mundo real, erros acontecem. Discos rígidos ficam cheios, redes caem e programas falham. É preciso testar como o sistema lida com esses tipos de problema forçando a ocorrência deles. Os casos de erro podem ser divididos em duas categorias: erros esperados e inesperados. Um erro esperado deve ser tratado pela implementação e deve ser considerado nos testes. Os erros inesperados são aqueles difíceis de prever ou muito trabalhosos para serem tratados. Esse tipo de erro normalmente leva a aplicação a encerrar o seu funcionamento. Nestes casos, os testes podem ser feitos para garantir que a aplicação será finalizada de uma maneira controlada.
- ***Checar por outros meios***. Normalmente existe mais de uma maneira de resolver um problema. A escolha de um algoritmo em detrimento de outros é feita devido ao melhor desempenho ou a uma característica desejável. O algoritmo selecionado é o que será utilizado em produção, mas isso não

significa que algum outro algoritmo não possa ser utilizado para fazer uma checagem do resultado esperado em um teste de unidade. Esta técnica é bastante válida quando se tem uma prova conhecida para um problema, mas sua performance deixa a desejar para ser utilizada no código que vai para produção.

- ***Usar a experiência***. Existem pessoas mais aptas a testar. Mesmo sem usar nenhum tipo de metodologia particular, elas têm uma habilidade nata para encontrar erros. Isso pode acontecer com a prática de testar e a experiência acumulada ao longo dos anos com o desenvolvimento de software. É difícil criar um procedimento para esse tipo de critério, já que é uma característica muito pessoal. O que pode ser feito nessas situações é enumerar uma lista das possíveis situações que levam a um erro e depois escrever casos de teste para cada uma delas.

O uso dessas estratégias não vai garantir que todos os erros sejam encontrados, mas é uma forma apurada de escolher casos de teste. Com o tempo e a experiência adquirida, pode-se montar um catálogo com os casos de teste mais vantajosos para cada situação, como o catálogo feito por Brian Marick em "*The Craft of Software Testing*"[28].

6.4.*Refactoring*

Em *eXtreme Programming*, o investimento no design do sistema deve ser feito todos os dias. Cada programador deve se empenhar para fazê-lo o melhor possível. Se o entendimento com relação ao melhor design não estiver claro, é preciso trabalhar gradualmente e de forma persistente para alinhar o design com aquilo que o programador considera bom (Beck & Andres, 2004).

Obter o melhor design na primeira vez em que se implementa uma solução, contudo, não é um fato comum. Um programador sempre sabe menos quando começa a escrever um software do que mais tarde, quando ele termina de implementá-lo (McConnell, 2004). Isto posto, é imprescindível o uso de uma técnica para melhorar o código conforme ele é escrito. "O *refactoring* é uma técnica para reestruturação do corpo de um código, alterando sua estrutura interna

[28] http:// www.testing.com/writings/short-catalog.pdf

sem modificar o seu comportamento externo." – Martin Fowler. Além disso, o *refactoring* é uma maneira disciplinada de limpar o código, minimizando as chances de introduzir novos *bugs*. Em poucas palavras, quando o *refactoring* é feito, o design do código é melhorado depois de ele já ter sido escrito (Fowler et al., 1999).

Por meio do *refactoring* pode-se partir de um design mal-feito – até mesmo um caos completo – e refazê-lo sob a forma de um código bem-feito. Basta mover um campo de uma classe para outra, isolar um pedaço de código fora de um método para fazer o seu próprio método ou deslocar o código para cima ou para baixo na hierarquia das classes. Cada passo é muito simples, mas o efeito cumulativo dessas pequenas mudanças pode melhorar radicalmente o design (Fowler et al., 1999).

Mas como perceber a necessidade *refactoring*? Martin Fowler definiu em (Fowler et al., 1999) alguns sinais de que o código que foi escrito é de má qualidade. A essas características, ele deu o nome de *code smell*. Alguns exemplos de *code smell* são: código duplicado; uma rotina muito longa; uma interface não oferece um nível consistente de abstração; uma lista de parâmetros possui muitos parâmetros; mudanças exigem alterações paralelas em várias classes; condicionais devem ser modificados em paralelo; uma rotina usa mais funcionalidades de outra classe do que da sua própria classe; um método tem um nome ruim; uma classe possui campos de dados públicos; comentários são utilizados para explicar um pedaço do código que é difícil de entender; e uso de variáveis globais. Quando um programador nota a presença de um *code smell*, ele deve iniciar o *refactoring*.

Esta evolução, ao mesmo tempo em que é uma oportunidade para o aperfeiçoamento, também pode se tornar um perigo. Cada programador deve se esforçar para escrever o melhor código possível tanto quando está começando a escrever uma nova funcionalidade quanto no momento em que está alterando um código existente. Mas só isso não é suficiente. É preciso que o código esteja resguardado por testes que assegurem o seu correto funcionamento. Caso contrário, não é possível afirmar que um *refactoring* não danificou o código, alterando um comportamento que antes funcionava perfeitamente.

Uma vantagem de utilizar o *refactoring* para desenvolver software é a divisão do trabalho do programador em duas atividades distintas: adicionar funcionalidades e fazer *refactoring*. Quando é preciso adicionar uma

funcionalidade, não se deve alterar o código que já existe. Esta é simplesmente uma tarefa de adicionar novas capacidades. O progresso é medido por meio dos testes que vão sendo escritos e, após a implementação das rotinas, passam. Quando é preciso fazer o *refactoring*, o propósito não é adicionar uma nova funcionalidade. Esta é apenas uma tarefa de reestruturação do código. Nenhum teste é adicionado (a menos que apareça um caso que não foi pensado anteriormente). Os testes só devem ser alterados quando é extremamente necessário, por exemplo, quando a assinatura de um método em uma interface é alterada.

Dessa forma, fica claro para o programador em qual momento se preocupar com implementar as novas funcionalidades e em qual momento melhorar – escrevendo de forma mais clara – aquilo que já está funcionando. Ao mesmo tempo, fazer *refactoring* melhora o design do software, faz com que o código torne-se mais fácil de entender e ajuda a encontrar *bugs*. Como conseqüência de todas essas vantagens, o uso do *refactoring* ajuda a programar mais rápido.

O *refactoring* é uma prática muito útil mesmo alheio ao uso de XP. Porém, existem alguns riscos quando feito constantemente. Fazer *refactoring* consome tempo e esforço e não existe um critério bem definido sobre o momento certo de parar de fazê-lo. Ao mesmo tempo, muitos *refactorings* alteram a interface de uma classe, o que normalmente gera trabalho para adequação do restante do código e dos testes. Um bom conselho a se considerar é "se não está quebrado, então não corrija" (Stephens & Rosenberg, 2003).

Além disso, mudar o código sempre que um design melhor for observado poderá fazer com que o sistema nunca fique pronto. Se o código funciona, tem um baixo (ou nenhum) número de defeitos e a arquitetura demonstra que o código está fundamentalmente completo, então não há necessidade alguma de modificar o código. Ele está pronto. O *refactoring* deve ser empregado com base na necessidade, ou seja, identificando o que é preciso para terminar o projeto.

Portanto, *refactoring* é uma parte integral do ciclo de desenvolvimento de software em XP. Os desenvolvedores alternam entre adicionar novos testes, novas funcionalidades e fazer o *refactoring* no código para melhorar a sua clareza e a sua consistência interna. Os testes de unidade automatizados asseguram que o *refactoring* não fez com que o código parasse de funcionar.

6.5.Mock Objects

Um aspecto fundamental dos testes de unidade é que estes devem testar apenas uma funcionalidade de cada vez. O código dos testes deve comunicar a intenção do programador da forma mais clara e simples possível. Isso pode ser difícil de acontecer se um teste precisa configurar várias dependências ou se o uso de um objeto nos testes apresentar efeitos colaterais. Pior do que isso, o código que está sendo testado pode depender de outro que não expõe as propriedades que determinam o estado necessário para a realização de um teste (Mackinnon et al., 2000). A maior parte dos problemas não triviais são complicados de testar isoladamente. Logo, torna-se improvável a criação de testes que não sejam complexos, incompletos, difíceis de manter ou interpretar.

Mock objects é uma técnica que propõe a substituição do código que define as funcionalidades por implementações falsas que emulam o código real. Em conjunto com o uso de interfaces, a utilização de *mock objects* nos testes de unidade melhora tanto o código da aplicação quanto dos casos de teste. Por meio destes, é possível que os testes sejam escritos para qualquer objeto, simplificando a estrutura dos testes e evitando a poluição do código em produção com elementos de teste (Mackinnon et al., 2000).

Os *mock objects* são passados por parâmetro para o código que está sendo testado. Isso permite que situações inesperadas e de falha possam ser reproduzidas com maior facilidade. Por exemplo, para testar uma falha de escrita em um arquivo podemos criar um *mock object* que estende a classe que escreve em arquivo. Cada teste de unidade pode configurar o *mock object* para falhar com uma exceção esperada e os desenvolvedores podem escrever testes para essa condição de erro específica. Essa prática é parecida com a escrita de *stubs*, com duas diferenças: 1) os testes podem ser feitos com um nível de granularidade mais fino do que o usual e 2) os *mock objects* podem verificar a sua consistência – por meio de instrumentação – apontando erros mais facilmente.

Deve-se notar que os *mock objects* não são usados para testar os objetos que estão sendo simulados. Eles são usados para testar um código que está para ser escrito ou já foi escrito e que depende destes objetos. Além disso, os *mock objects* não devem reimplementar as funcionalidades que estão sendo emuladas. Eles devem apenas reproduzir as resposta necessárias para a realização de um teste em

particular. Por isso, a maioria dos métodos de um *mock object* simplesmente não faz nada ou apenas armazena valores em propriedades locais (Mackinnon et al., 2000).

O uso de *mock objects* é uma excelente técnica para revelar a interface de objetos dos quais o código que está sendo testado depende. Cada *mock object* é uma hipótese do que o código real eventualmente pode vir a fazer. Quando o código que está sendo testado se consolida, os *mock objects* possuem características que podem ser extraídas de forma a definir novas interfaces que o sistema deve implementar.

Mock objects é uma prática complementar que, se usada de forma imprudente, pode criar uma confusão de objetos falsos. Quando é preciso configurar muitos destes objetos e estes dependem de outros *mock objects*, é sinal de que os testes de unidade não estão bem delimitados. Por conseguinte, quando esta técnica é usada, apenas os testes e o código em produção devem ser reais (Mackinnon et al., 2000).

Deste modo, *mock objects* são um acessório que simplifica o desenvolvimento dos casos de teste. Com o uso destes, evita-se que configurações complexas tenham que ser feitas para testar uma característica específica do código que está sendo produzido. Além disso, estes ajudam na definição de objetos que ainda não foram criados, mas que serão necessários para o funcionamento do sistema como um todo. Também ajudam a testar situações inesperadas ou difíceis de reproduzir. Tudo isso é fundamental para manter o programador focado na funcionalidade que realmente precisa ser implementada e testada.

6.6.Ferramentas

"Sem o uso de ferramentas adequadas, TDD é praticamente impossível" (Ambler, 2003b). A aplicação da técnica de *Test Driven Development* torna-se inviável sem a utilização de boas ferramentas. Nesta seção estão algumas indispensáveis para um bom aproveitamento da prática do TDD.

6.6.1.JUnit[29]

Para que o desenvolvimento dirigido por testes fosse possível, era necessária uma ferramenta básica que permitisse escrever e executar casos de teste. A primeira implementação desse tipo de ferramenta foi feita em Smalltalk e foi chamada de SUnit. A partir dela, abstraiu-se uma arquitetura conhecida como "*framework* xUnit". "Esta arquitetura foi implementada em várias linguagens de programação e para diferentes plataformas" (Murphy, 2005).

O JUnit é uma implementação do *framework* xUnit feito na linguagem Java que permite escrever testes de unidade e executá-los repetidas vezes. Entre as suas funcionalidades destacam-se: o uso de assertivas para testar resultados esperados; a possibilidade de criação de objetos comuns que são compartilhados por todos os testes; e os *suites* de teste para organização e execução conjunta de vários testes.

```
import junit.framework.TestCase;

public class ExemploTest extends TestCase {

    /**
     * Tests emptying the cart.
     */
    public void testSoma() {

        int resultado = Calculadora.soma( 2, 2 );

        assertEquals( 4,  resultado );
    }
}
```

Acima, um exemplo simples de um teste de unidade feito com o JUnit. Como pode ser visto, é preciso estender a classe `TestCase` e escrever métodos com o prefixo `test`. Os resultados são verificados usando métodos especiais como o `assertEquals`. Neste exemplo, é feito um teste para verificar que o método estático `soma` da classe `Calculadora` retorna 4 quando recebe os parâmetros 2 e 2.

6.6.2.TestNG[30]

Mais do que qualquer outro *framework* de teste, o JUnit levou os desenvolvedores a entenderem a necessidade de testar ou, mais especificamente, de fazer testes de unidade. Porém, com a evolução da linguagem Java, o JUnit se

[29] http://www.junit.org/

[30] http://testng.org/

tornou simples demais. O TestNG foi feito para aproveitar as novas capacidades da linguagem Java. Inspirado no JUnit e também no NUnit, um *framework* para testes de unidade em .Net, o TestNG introduziu novas características aos testes de unidade, como por exemplo:

- Suporte a anotações em Java.
- Configurações de teste em arquivos XML.
- Não requer a extensão de uma classe.
- Permite definir grupos de teste.
- Execução de testes em paralelo.
- Aceita parâmetros nos métodos de teste.

A versão 4.0 do JUnit, lançada há pouco tempo, também implementa algumas das melhorias descritas acima.

6.6.3.Eclipse[31]

O Eclipse é um ambiente de desenvolvimento integrado (IDE) que possui um conjunto de ferramentas que facilitam a utilização do TDD. Principalmente no desenvolvimento Java, o Eclipse oferece funcionalidades para *refactoring* de código, integração direta com o JUnit, assistentes e modelos para a criação de classes, interfaces e casos de teste entre outras funcionalidades vitais para o ciclo de desenvolvimento de software.

Uma das características do desenvolvimento dirigido por testes é a necessidade constante de *refactoring*. Muitos dos *refactorings* estão relacionados com alterações que quebram o restante do código que depende do fragmento que está sendo melhorado. Um *refactoring* simples como "Renomear um Método" está profundamente relacionado com a mudança da assinatura de um método. Fazer esse tipo de modificação sem a garantia de que ela será propagada para todas as classes que referenciam o método é muito ruim. Com o Eclipse, essa transformação (e muitas outras) pode ser feita com facilidade, sem quebrar o código já existente, pois a IDE propaga automaticamente as alterações para o restante do código.

[31] http://www.eclipse.org/

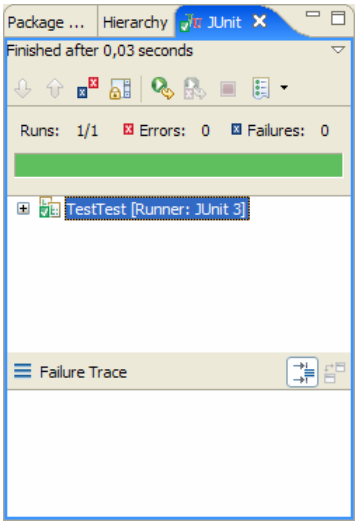

Figura 11: Barra verde, sucesso na execução dos testes no Eclipse.

Além disso, TDD implica executar os casos de teste freqüentemente. O Eclipse automaticamente compila o código e os testes enquanto as alterações são feitas e permite rodar todos os testes com o clique de um botão. A Figura 11 mostra uma visão do Eclipse com o resultado da execução de um conjunto de testes. A barra fica verde em caso de sucesso e vermelha em caso de falha.

6.6.4.IntelliJ IDEA[32]

O IntelliJ IDEA é outro ambiente de desenvolvimento integrado para desenvolvimento de aplicações Java. Concorrente direto do Eclipse, também possui suporte ao *framework* JUnit e vários mecanismos para facilitar o *refactoring* de código. Uma diferença clara entre as duas ferramentas é que o Eclipse é gratuito e *open source*, enquanto o IntelliJ IDEA é uma ferramenta proprietária.

6.6.5.EasyMock[33]

Escrever e manter *mock objects* é uma tarefa muitas vezes maçante e que pode introduzir diversos erros. EasyMock é uma biblioteca que permite criar *mock objects* dinamicamente a partir de interfaces Java – durante a execução dos testes. Devido à forma como o EasyMock registra as expectativas, a maioria dos

[32] http://www.jetbrains.com/idea/

[33] http://www.easymock.org/

refactorings não afeta os *mock objects*. Nenhum *mock object* é realmente escrito e nenhum código é gerado. Por isso, nenhum código precisa ser modificado.

EasyMock é uma combinação perfeita para o *Test Driven Development*.

6.7.Extensões do JUnit

Existem algumas situações em que apenas o JUnit não é suficiente para testar determinados comportamentos. Como testar aplicações Web? E aplicações que se comunicam com um banco de dados? Uma das vantagens de usar o JUnit é que este possui uma variedade de extensões para resolver cada tipo de problema.

6.7.1.DBUnit[34]

O DBUnit é uma extensão do JUnit para projetos em que é preciso se comunicar com um banco de dados. Esta ferramenta permite, entre outras funcionalidades, colocar o banco de dados em um estado conhecido durante os testes. Esta é uma ótima maneira de evitar a grande quantidade de problemas que podem ocorrer quando um teste corrompe o banco de dados e faz com que os testes subseqüentes falhem ou acentuem o problema.

O DBUnit tem a habilidade de exportar e importar as informações do banco de dados no formato XML. Além disso, também ajuda a verificar se as informações do banco de dados são iguais ao conjunto de valores esperados.

6.7.2.XMLUnit[35]

O XMLUnit é uma extensão do JUnit que permite utilizar assertivas para verificação da estrutura e o conteúdo de arquivos XML. Faz parte de um projeto aberto hospedado no sourceforge.net. Com a utilização em larga escala de XML para troca de mensagens, configuração e os mais diversos requisitos nos sistemas atuais, a utilização do XMLUnit facilita a verificação com relação ao que está realmente sendo produzido.

[34] http://dbunit.sourceforge.net/

[35] http://xmlunit.sourceforge.net/

6.7.3.HttpUnit[36]

O desenvolvimento dirigido por testes implica chamadas diretas ao código que está sendo testado. Mas como fazer para testar aplicações Web? O HttpUnit é uma extensão do JUnit que torna possível testar aplicações desse tipo. Esta ferramenta pode emular um *browser*, tratar de *frames*, *cookies*, redirecionamentos e etc. Também permite ver páginas como texto puro, como XML DOM ou como uma coleção de objetos que representam *links*, *frames*, imagens, entre outros.

Ao contrário da maioria das ferramentas comerciais, o HttpUnit não se baseia em gravar e reproduzir situações. Sua API permite que um programador defina o que quer ver ou alterar, mesmo antes de o site estar pronto. Logo, esta é uma ferramenta essencial para o desenvolvimento de aplicações Web baseadas em testes.

6.8.Precauções

Usar *Test Driven Development* implica, inicialmente, escrever casos de teste. Testes automatizados bem feitos provavelmente dobram a quantidade de código escrito em um projeto. Por isso, assim como o código-fonte, os testes também precisam ser gerenciados. Para fazer isso, recomenda-se o uso de algumas regras:

- Os testes devem ficar em um diretório separado, organizados sob a forma de árvores paralelas. Ou seja, as classes de negócio e de teste ficam no mesmo pacote, porém em diretórios diferentes. Esta estrutura é a mais limpa e não restringe o acesso que a classe de teste precisa ter com relação à classe que está sendo testada. Esta também é a estrutura para organização de código adotada por ferramentas de *build* como o Maven (Massol & Van Zyl, 2006).
- Um arquivo de teste deve ser escrito para cada classe. Os casos de teste devem ser implementados para cada método que possivelmente possa falhar.

[36] http://httpunit.sourceforge.net/

- Os testes devem estar organizados de forma que possam ser executados individualmente, em conjunto por arquivo, por projeto ou para o sistema inteiro.
- O *build* e a execução dos testes precisam ser automatizados. A inspeção dos resultados também. Assim é possível executá-los tantas vezes quanto for necessário para validar e revalidar um sistema.
- Uma ferramenta de *build* deve ser configurada para executar os testes automaticamente. Todos os testes de unidade devem passar antes de gerar uma nova versão do sistema. Algumas ferramentas, como o Maven, fazem isso por padrão (Massol & Van Zyl, 2006).
- Assim como o código-fonte, os testes também devem estar no controle de versões do projeto.

Quando uma organização compreensível estiver pronta para os testes, estes podem ser escritos. Nesta tarefa, no entanto, cada programador deve levar em consideração algumas características determinantes para um bom uso do TDD.

Existe uma característica chave implícita nesta forma de desenvolver: ***Os testes têm que ser ágeis***. Ou seja, precisam ser simples e rápidos de executar. Não será possível rodá-los a cada minuto se eles levarem meia-hora para executar por completo. Logo, não é em um teste de unidade o lugar para longos testes de performance e carga ou para testes que enumeram todas as possíveis combinações de entrada que uma rotina pode receber (Thomas & Hunt, 2002).

Nunca escrever testes manuais (Crispin & Rosenthal, 2002) ou que dependam de intervenção humana. Um teste manual não apresenta as qualidades de um bom caso de teste. Por isso, ***todos os testes de unidade devem ser automatizados***. Outras formas de teste (como testes de aceitação, testes de integração e testes funcionais) também devem ser automatizadas. Evidentemente, em algumas situações, como por exemplo, fazer um teste de aceitação automatizado baseado na interface com o usuário, pode ser complicado ou caro demais. Nesses casos, é melhor ter um roteiro de um caso de teste manual para seguir do que não ter nenhum teste. De qualquer forma, deve-se procurar sempre por formas de automatizar todos os testes que estão sendo feitos.

As APIs públicas devem ser documentadas. Uma das regras de *refactoring* diz que comentários no código devem ser substituídos por código mais fácil de

entender. Isto é verdade, porém o fato de o código ser limpo e claro não significa que nenhuma documentação deva ser escrita. Apesar de os desenvolvedores poderem se basear nos testes de unidade para determinar como usar um método de uma classe, uma documentação clara da API também é muito útil. Comentários bem escritos usando o estilo Javadoc para cada método ou propriedade pública de uma classe são muito úteis (McBreen, 2002).

Já que escrever testes para todos os casos é uma tarefa praticamente impossível, a arte de testar está em determinar os casos de teste que podem encontrar mais erros (McConnell, 2004). O programador deve eliminar aqueles que não expõem nada de novo e ***se concentrar nas situações que podem gerar respostas diferentes*** – ou condições limite – ao invés de se concentrar nas situações que geram o mesmo resultado. Identificar essas condições limite é uma das partes mais significativas de um teste de unidade. Normalmente, é nessas situações que a maioria dos *bugs* se encontra – nas extremidades. Para facilitar o pensamento sobre condições limite, (Hunt & Thomas, 2003) definiram o acrônimo CORRECT:

- *Conformance* (conformidade): O valor está de acordo com o formato esperado?
- *Ordering* (ordenação): O conjunto de valores está ordenado ou desordenado apropriadamente?
- *Range* (abrangência): O valor está dentro de uma faixa mínima ou máxima de valores?
- *Reference* (referência): O código referencia algo externo e que não está sob o controle direto do próprio código?
- *Existence* (existência): O valor existe (por exemplo, é não-nulo ou diferente de zero)?
- *Cardinality* (cardinalidade): Existem valores em quantidade suficiente?
- *Time* (tempo): Tudo está acontecendo na ordem esperada? No tempo certo? Em tempo?

Para cada um desses itens, é preciso considerar se alguma das condições pode existir no método que está sendo testado e o que deve acontecer se qualquer uma delas for violada.

Os testes também devem ser bem feitos. Um dos resultados do uso do *Test Driven Development* é a garantia de que o código que está sendo gerado está limpo e funciona. Mas e se os testes para esse código estiverem errados? Ou se forem mal-feitos? Apenas seguir o ciclo de desenvolvimento proposto pelo TDD não é garantia de que o código funciona como o esperado. Os testes devem ter algum significado no contexto de funcionalidades esperadas do sistema. Além disso, para identificar problemas no código dos testes, o desenvolvedor deve procurar por *test smells* (análogos aos *code smells* utilizados durante o *refactoring*). Meszaros (2007) e Deursen et al. (2001) definiram alguns *test smells* e formas de melhorar o código dos testes com problemas.

Mesmo sabendo de todas as vantagens que esta prática propicia, muitas pessoas continuam achando pretextos para não testar software. A seguir, uma lista das desculpas mais comuns utilizadas por pessoas que não se predispõem a testar, seguidas de uma explicação de por que elas não são verdadeiras:

- ***"Escrever testes leva muito tempo"***: Programadores não se sentem produtivos a menos que estejam codificando algo que será executado na aplicação final (Thomas & Hunt, 2002). Na realidade, esta é uma visão um tanto restrita do processo de desenvolvimento de software. Quanto tempo é gasto com depuração de código? Quanto tempo é gasto reescrevendo código que parecia estar funcionando, mas no fundo possuía *bugs*? Quanto tempo é necessário para localizar e isolar, no código-fonte, um *bug* que foi reportado? Quanto tempo se perde com mudanças por causa de um requisito mal-definido? O uso de testes tem um custo. O esforço para desenvolver código é maior durante todo o processo. Porém, aumenta também a produtividade dos desenvolvedores, pois não têm que perder tempo extra pouco antes do término do projeto corrigindo problemas que não foram capturados mais cedo, durante o desenvolvimento.
- ***"É muito demorado executar os testes"***: A maioria dos testes leva poucos segundos para serem executados. Além disso, testes que são executados lentamente podem ser configurados para rodar com menos freqüência (apenas quando o desenvolvedor quiser).
- ***"Não é meu trabalho testar o código que escrevo"***: Então qual é mesmo o trabalho de um desenvolvedor? Se considerarmos que a função de um programador é fazer código que funcione corretamente, então, assegurar

que o código que está sendo produzido funciona como o esperado também é tarefa do programador.

- ***"Escrever testes tira a minha concentração em escrever o código"***: Alguns desenvolvedores acreditam que o processo incremental de desenvolvimento dirigido por testes vai atrapalhar o fluxo de trabalho deles. Esta dificuldade advém do fato de que os programadores não estão acostumados a pensar dessa maneira. Por isso, não parece espontâneo em princípio (Thomas & Hunt, 2002). Mas com o tempo, o constante ciclo de teste e codificação apenas reforça a maneira como cada programador pensa nas soluções.
- ***"Eu não tenho certeza quanto ao comportamento do código, por isso eu não posso testá-lo"***: Se realmente essa dúvida existir, então talvez não seja hora de escrevê-lo. Talvez seja melhor começar com um protótipo para esclarecer as idéias com relação aos requisitos que devem ser atendidos.
- ***"Mas o código está compilando"***: Compiladores e interpretadores são capazes de verificar a validade da sintaxe do código que está sendo escrito. Eles não podem verificar, entretanto, a semântica do código.
- ***"Eu não preciso de testes, pois tenho certeza de que meu código está correto"***: Até mesmo os melhores programadores cometem erros. Esta abordagem pode até funcionar para projetos pequenos, de pouca importância e onde apenas uma pessoa mexe no código. Mas quando o número de pessoas aumenta e o erro de uma começa a atrapalhar as outras, não ter um mecanismo para encontrar *bugs* torna muito difícil o diagnóstico dos problemas.
- ***"Mas estou sendo pago para escrever código e não para escrever testes"***: Usando a mesma lógica, um programador não é pago para passar horas na frente de um depurador caçando erros. Testes de unidade são como um ferramenta para auxílio no desenvolvimento de software, assim como um editor, uma IDE e um compilador.
- ***"Testar é muito complicado"***: Alguns desenvolvedores não escrevem testes de unidade porque já estão no limite. Eles escrevem código na base do acréscimo: adicionam uma funcionalidade aqui, consertam um defeito ali, sem nunca saber quando eles realmente terminaram. Por isso, eles

sentem que se tiverem que escrever os testes em adição ao código, sua produtividade vai cair (Thomas & Hunt, 2002). Usualmente estas são as pessoas que mais se beneficiam com o uso de TDD, pois esta prática permite que os problemas complicados sejam resolvidos paulatinamente.

- ***"O código que eu estou escrevendo não pode ser testado"***: Essa é a única desculpa que deve ser levada em consideração. Existem alguns casos conhecidos onde o uso de testes de unidade não é aplicável. Em particular, testes de unidade não se ajustam bem a sistemas *multithread* (por exemplo, o tratamento de eventos em Swing), com troca assíncrona de mensagens (por exemplo, a troca de mensagens em aplicações J2EE) ou não-determinísticos (por exemplo, algoritmos como caixeiro viajante, complexos de serem provados que estão corretos). Isto, no entanto, não é desculpa para não fazer testes de uma vez por todas. Todo software deve ser projetado para ser pouco acoplado – esta é uma boa prática de programação – e por isso, os testes de unidade devem ser feitos para tudo aquilo que for possível. Em muitos dos casos, o uso de *mock objects* e extensões do JUnit podem facilitar testes a primeira vista impossíveis.

Testes escritos antes do código preocupam-se apenas com uma visão reduzida do sistema como um todo (Beck & Andres, 2004). Isso é uma deficiência, pois não é possível saber se a classe que está sendo testada funciona corretamente com os outros objetos do sistema. Além disso, testes de unidade só pegam os erros que foram antecipados pelo programador (Stephens & Rosenberg, 2003). Ou seja, é inevitável que ele cometa um erro ao escrever um caso de teste ou simplesmente não perceba uma circunstância específica.

Por isso, é preciso fazer outros testes. *Test Driven Development* é primariamente uma prática de análise e design que tem como efeito colateral a produção de código com uma alta cobertura de testes. Porém, existe muito mais teste além dos testes de unidade (Ambler, 2003a). Ainda é preciso considerar outros tipos de testes, como testes de aceitação, de integração, funcionais e do sistema, por exemplo. Cada tipo de teste tem uma característica específica e será capaz de identificar problemas que não podem ser percebidos por outras abordagens.

Outro problema sério que o mau uso do TDD pode criar é o efeito programadores "burros" ou "preguiçosos". Isso significa que, ao invés de estudar

sobre um assunto para pensar na melhor solução, os programadores podem ficar fazendo testes de tentativa e erro até chegar a uma solução funcional. Esse não é o resultado que se espera com o uso desta técnica. Esta deve servir como um auxílio na forma de se pensar no sistema que está sendo produzido. Isso não significa que um entendimento sobre o problema a ser resolvido seja dispensável.

Resumidamente, TDD sozinho é capaz de melhorar a qualidade do software que é desenvolvido. No entanto, para obter vantagens como desenvolvimento incremental e contínuo feedback do cliente, é preciso que práticas como *build* automatizado e integração contínua estejam funcionando de forma auxiliar. Até mesmo a presença de obstáculos não é um pretexto para negligenciar esta técnica. Tomando os devidos cuidados, TDD é uma prática indispensável para o desenvolvimento de software em geral.