Qualidade interna e externa de um software e seu código: Como garantir a entrega de um projeto de software com um bom nível de manutenibilidade

Na vasta maioria de projetos de software requisitados para fornecedores internos (o próprio departamente de TI) ou externos (consultorias e fábricas de software), podemos notar que os principais itens de performance medidos são o prazo de entrega e o custo. Conforme nos mostra Robert Austin magistralmente, no seu livro "Measuring and Managing Performance in Organizations", medir um sistema complexo através de poucas métricas o torna disfuncional e gera efeitos contrários aos pretendidos.

Quando se mede a performance de um projeto de software apenas por seu custo e prazo nota-se um comportamento disfuncional clássico: como o gerente de projeto e os membros da equipe só serão medidos pela velocidade de entrega, então tudo é feito às pressas e usualmente sem a atenção devida a um bom design, a testes unitários e funcionais e a refatorações constantes. Sem esses cuidados, o sistema costuma entrar com um número de defeitos acima do desejado e ainda se torna um código legado desde sua entrada em produção. Essa medição de performance também faz com que os níveis gerenciais demandem mais de 40 horas semanais dos profissionais (muitas vezes sem o devido pagamento de horas extras), o que ainda gera uma degradação de moral da equipe e um turn-over mais alto na empresa.

Como então podemos obter um bom código, que não vire legado assim que é entregue para a equipe que irá mantê-lo? Primeiro vamos definir o que é um código legado, o que é um código bom e o que significa o requisito não-funcional de manutenibilidade.

Quando se fala a palavra código legado, na cabeça dos desenvolvedores surge uma visão de código impossível de entender, estruturalmente complexo, cheio de condições encadeadas, código macarrônico e métodos com centenas e até milhares de linhas de código! Segundo Michael Feathers (em seu livro "Working Effectively with Legacy Code") código legado é aquele difícil de entender e, consequentemente, difícil de alterar para implementar novas funcionalidades ou corrigir defeitos.

Mas, segundo ele, a definição mais importante é**: código legado é código sem testes unitários automatizados**. Essa definição radical e inusitada nos mostra a importância de uma suíte de testes unitários para manter a sanidade do código e nossa coragem de fazer modificações quando necessárias. Sem testes os desenvolvedores adotam a postura de não mexer em código que "não é deles", por medo de gerar defeitos.

Também não fazem modificações estruturais para manter o código coeso e com baixo acoplamento, especialmente quando incluem novas funcionalidade. Portanto, sem testes unitários é difícil dar manutenção evolutiva e corretiva no código e mantê-lo bom, evitando que se torne legado e complexo.

Nossa próxima definição é sobre código bom. Essa é uma definição mais complexa. Gosto muito do termo inventado por Kent Beck: "code smell". O "mau cheiro" em código é um sintoma de que algo está errado. É uma indicação de que o código precisa ser refatorado ou o design deve ser reexaminado. Alguns "maus cheiros" clássicos são (acesse a taxonomia de maus cheiros, caso queira mais detalhes):

> **Métodos longos -** esse tamanho pode variar, mas algumas pessoas consideram que um método com mais de 50 linhas de código já é um método longo (alguns mais puristas chegam a dizer que um método deve ter no máximo 10 linhas!). Veremos adiante no artigo que a complexidade ciclomática pode nos ajudar a obter um indicador mais preciso.

Classes grandes e complexas - normalmente as que não seguiram o princípio de alta coesão Classes com intimidade com muitas outras classes - as que não seguiram o princípio de baixo acoplamento Classes preguiçosas – que não fazem muita coisa ou são executadas com pouca freqüência

Código duplicado – essa é uma das maiores pragas que devem ser combatidas. E o requisito não-funcional da manutenibilidade? Bom, esse é, como qualquer requisito não-funcional, fundamental para uma definição da arquitetura de um software. É a facilidade de manutenção para inclusão de novas funcionalidades, correção de defeitos e alterações de regras de negócio. Esse requisito é fundamental para qualquer sistema que venha a ter uma vida útil longa em produção e que sofrerá contínuas modificações. Clientes e outros stakeholders não costumam falar claramente "esse sistema deve durar 10 anos em nossa empresa, é crítico e terá um número grande de requisitos novos mensalmente". Por isso a manutenibilidade deve ser questionada claramente para os stakeholders. Alguns dizem que não se deve preocupar com manutenibilidade se o código durar pouco tempo. Tome cuidado, porque a tendência de todo código criado é a de continuar sendo usado por tempos razoáveis. Portanto, desconfie se disserem que o sistema ficará pouco tempo em produção. Nem sempre o cliente sabe tudo (vide o livro "O Dilema da Inovação" de Clayton Christensen para detalhes desse fenômeno no mundo dos negócios) !

Agora o problema: tendo em vista que sabemos a importância de se obter um código bom, não legado e com alta manutenibilidade, como medir isso em um fornecedor (interno e/ou externo) e como garantir que essa qualidade existe? Aí vão algumas dicas.

A primeira parte é a mais fácil: medições de código. Hoje temos várias ferramentas (neste artigo estou focando em ferramentas para Java, mas saibam que também existem similares para .NET e outras plataformas) que ajudam em 80% do esforço de detectar problemas. Vamos a algumas delas para exemplificar:

- ✓ Usar JUnit para fazer os testes unitários automatizados e, desse modo, não deixar o código virar legado conforme definição do Feathers. Como um cliente pode medir se os testes unitários estão sendo feitos e se são relevantes? Uma forma muito boa é medir a cobertura de código (de linhas e de desvios) realizada pelos testes unitários. Para isso é possível usar uma ferramenta open source como o Cobertura ou uma ferramenta como a embutida no Rational Application Developer.

- ✓ Usar uma ferramenta como o JavaNCSS para contar a quantidade de linhas de código e a complexidade ciclomática por método. Com essas métricas pode-se analisar se o código de cada método está complexo demais e se deve ser quebrado para manter a coesão. É possível usar uma ferramenta como o crap4J para tirar métricas combinadas entre cobertura de código e complexidade ciclomática.

- ✓ Usar uma ferramenta de análise estática de código como o PMD ou a embutida no Rational Application Developer para avaliar regras básicas de código. Essa é uma ferramenta essencial e que deve ser utilizada antes de qualquer inspeção formal feita por desenvolvedores. Ela irá pegar erros básicos que forem configurados. Ela detecta falhas como:
    - o Métodos longos

- Nomes de variáveis e métodos curtos
- Nomes de variáveis que não começam com letra minúscula
- Identação de chaves
- Quantidade excessiva de parâmetros
- Acoplamento entre objetos
- Aninhamentos grandes de ifs e elses
- Densidade de switch
- Código não utilizado (variáveis, parâmetros, métodos, etc)

✓ Usar a ferramenta CPD (Copy/Paste Detector) para identificar código duplicado e passível de refatoração. Usar uma ferramenta de análise de dependências de código orientado a objeto como o JDepend ou o Rational Software Architect (com sua capacidade de descoberta arquitetural e detecção de antipatterns arquiteturais).

Todas as ferramentas acima podem ser executadas através de um simples comando do Ant, após feita a configuração correta. Isso pode ser ainda colocado para executar dentro de um servidor de integração contínua, que pode enviar relatórios diários por email do status das métricas de código.

Além de medir o código usando ferramentas, sempre faça auditorias em cada uma das versões entregues (lembre que ferramentas garantem 80% dos casos, normalmente os mais típicos). As ferramentas não excluem o uso de bons desenvolvedores inspecionando formalmente código. Segundo McConnell, no livro “Code Complete”, de 45 a 70% dos defeitos de um código são removidos quando este passa por uma inspeção formal de desenvolvedores experientes.

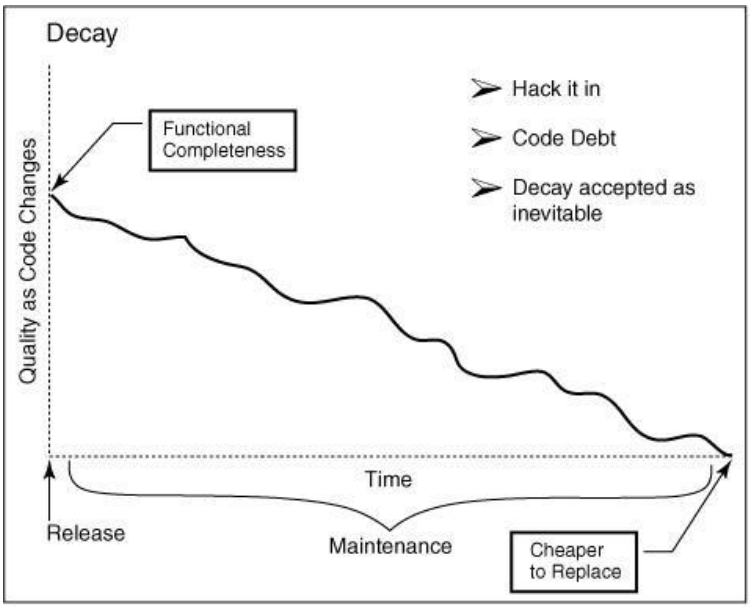
Agora vamos à parte que dará mais trabalho (em termos de negociação e possíveis conflitos): como garantir essa qualidade de seu fornecedor? A resposta: estabeleça um SLA (Acordo de nível de serviço) para cada parâmetro importante discutido na primeira parte (métricas). Pode-se definir, por exemplo:

- Uma complexidade ciclomática de até 15 como aceitável, de 16 a 30 como aceitável se o fornecedor justificar de forma admissível ou inaceitável quando maior que 30.
- Uma cobertura de linhas de código de 80% como aceitável. Abaixo disso é inaceitável e acima disso é bônus.
- Regras definidas pelo cliente nas ferramentas de análise estática de código devem ser todas seguidas. Não deve aparecer nenhum erro.

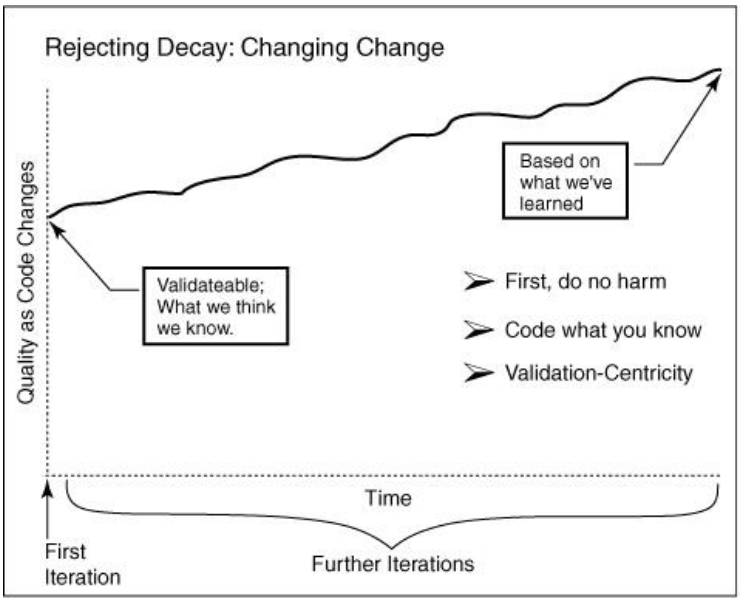
Com base nisso pode-se elaborar penalidades no caso do fornecedor não estar dentro do SLA e bônus por ultrapassar os limites aceitáveis do SLA. È possível também elaborar SLAs para que o fornecedor realize testes funcionais automatizados e testes de performance automatizados.

Testes automatizados são mais garantidos porque podem ser executados para confirmar se foram feitos (diferentemente de casos de testes manuais) e ainda se tornarão um ativo de sua empresa, pois serão usados como testes regressivos. Um mínimo de testes funcionais automatizados (nem que sejam apenas os chamados "smoke tests") deve ser requerido.

Com essas técnicas você pode sair da situação da figura abaixo:



Para entrar numa situação bem melhor como essa:



E uma dica: se o seu fornecedor questionar e dizer que terá que aumentar o custo por causa disso, pergunte a ele: como você fazia então para garantir a qualidade e manutenibilidade de seu código sem essas métricas? Ou não fazia? Provavelmente o fornecedor que faz essa colocação não terá uma boa resposta verdadeira.

Outra dica: se o seu fornecedor se diz “agile”, não aceite apenas que haja entregas durante iterações curtas. Isso é apenas um lado (também importante). Exija a qualidade acima para realmente provar a agilidade do fornecedor. Processos ágeis primam muito pela qualidade interna e externa do código. Esse aviso é importante pois já

pode-se notar no mercado o que costumam chamar de empresas “pseudo-agile”. São aquelas que dizem que praticam Scrum e XP, mas na verdade praticam mesmo a codificação caótica e indisciplinada, tudo que processos ágeis não são.

Por que estou dando essas informações valiosas? Porque acredito que, se os clientes foram mais exigentes com seus fornecedores de software, então todo o país terá a ganhar. Formaremos profissionais de maior qualidade, que geram um produto (código) de maior qualidade e aí poderemos competir no mercado externo com outros profissionais não apenas nos termos de custo e sim da qualidade do trabalho. Mudaremos de uma estratégia de oceano vermelho de custo para uma estratégia de oceano azul da diferenciação.

Portanto relembrando para o cliente: elabore contratos que demandem entregas curtas (desenvolvimento iterativo) com qualidade interna e externa medida por ferramentas, pessoas e SLAs. Vamos melhorar juntos o nível de qualidade dos nossos produtos!

Dando uma dica para fornecedores: Sejam pró-ativos e ofereçam esse tipo de contrato para seus clientes. Mudem suas estratégias de desenvolvimento e ofereçam qualidade com agilidade. Gerem benefícios de valor agregado, antes que seus concorrentes o façam!