Hábitos de Qualidade para Programadores

Como fazer software de qualidade sem processos, com profissionais liberais em pequenas empresas

Fábio Ferreira de Souza – RA 612100517

Universidade Nove de Julho

São Paulo, Brasil

mpc.fabio@gmail.com

***Resumo** – Para garantir a qualidade, a engenharia de software propõe o estudo e uso de processos de qualidade que podem ser aplicados por qualquer programador, independentemente do tipo de projeto. Por meio de mais conhecimento sobre os princípios de qualidade, (validar, quantificar, monitorar, planejar a qualidade de um software) é possível mudar os hábitos pessoais do programador quando este entende os pontos comuns de qualidade que surgiram desde o início com o PMI e CMMI, e que existem também em metodologias ágeis. Estes princípios se tornaram valores pilares das ISO, até o surgimento do MPS.br. Assim da mesma forma que estes valores fundamentais de qualidade se transformaram em processos, é também possível transformá-los em hábitos de qualidade pessoais para o programador aplicar em qualquer projeto, agregando valor, garantindo o custo e o prazo de entrega.*

***Palavras-chave:** hábitos de qualidade, qualidade de software, princípios de qualidade.*

***Abstract** – To ensure quality, software engineering of proposes the study and use of quality processes that can be applied by any developer, regardless of the type of the project. Through more knowledge about quality principles, (to validate, quantify, monitor, plan the quality of the software) you can change personal habits of the developer when one considers the common points of quality that were there since the beginning with PMI and CMMI, and which are also featured in agile methodologies. These principles have become pillars of ISO values, until the creation of MPS.Br. In the same way that these fundamental values of quality became processes, you can also turn them into personal habits of quality so that the developer can implement any project, adding value, ensuring the cost and the delivery schedule.*

***Key words:** habits of quality, quality of the software, principles of quality.*

I. INTRODUÇÃO

A qualidade é um valor subjetivo, mas com as normas e padrões é possível quantificar a qualidade por meio de métricas e assim conseguir fazer comparações justas para qualificar o que é bom ou não.

Os processos de qualidade podem garantir a qualidade desde que as pessoas envolvidas sigam estes processos, mas não é isso que acontece em projetos pequenos quando os processos não são aplicados, mesmo sendo feitos pelo mesmo programador que segue processos em outros

projetos maiores, pois a qualidade deveria ser um hábito do programador e não apenas um item a ser cumprido nas listas de tarefas de um processo.

A responsabilidade da qualidade está somente no programador quando não existe um processo definido, por isso este deve ter conhecimento das causas e efeitos de se ter hábitos de qualidades independentemente de processos.

De acordo com o dicionário online português [1], hábito significa: “comportamento que determinada pessoa aprende e repete frequentemente, sem pensar como deve executá-lo. Uso, costume; maneira de viver; modo constante de comportar-se, de agir: desde criança adquirira o hábito de ler deitado. Muitas ações da vida cotidiana constituem hábitos. Imagine como seria difícil alguém descer uma rua se tivesse que pensar em cada um dos atos necessários a dar cada passo. O hábito difere do instinto, que é um comportamento inato, não aprendido”

O programador também precisa conhecer a si mesmo como indivíduo, não somente as linguagens de programação, para que também possa se reprogramar e aperfeiçoar-se; Para isso conhecer o processo psicológico de formação de hábitos para criar e modificar hábitos cotidianos, e também conhecer as principais características comuns dos padrões de qualidade de software e como aplicar isso pode ser benéfico para si e consequentemente para os seus projetos.

II. HISTÓRIA DOS PADRÕES DE QUALIDADE DE SOFTWARE

Analisando a tabela 1 vemos os anos de lançamentos dos principais documentos que são usados hoje como normas e padrões de processo de qualidade de software.

Ano	PMI	CMMI	ISO 15504	ISO 9126	MPS.br
1969	Fundação do Project Management Institute				
1983	White Paper				
1987		CMM			
1993			SPICE		
1994					
1995			ISO 12207		
1996	1ªed			NBR 13596	
2000	2ªed				
2001				Início	
2002		1.1	12207 rev		
2003			Parte 2	Partes 1 a 4	
2004	3ªed		Parte 1, 3, 4		
2005					Início
2006		1.2			1.1
2007					1.2
2008	4ªed		TR-6, TR-7		
2009					
2010		1.3			
2011			TR-9		
2012					
2013	5ªed				

Tabela 1- Quadro histórico dos principais padrões de qualidade

Percebe-se que no Brasil o padrão de qualidade de software MPS.br não tem nem 10 anos, e a cerca 20 anos só existiam conceitos muito genéricos do PMI sobre formas de medir processos de qualidade, onde a produção de software poderia seguir estes mesmos modelos a fim garantir alguma qualidade.

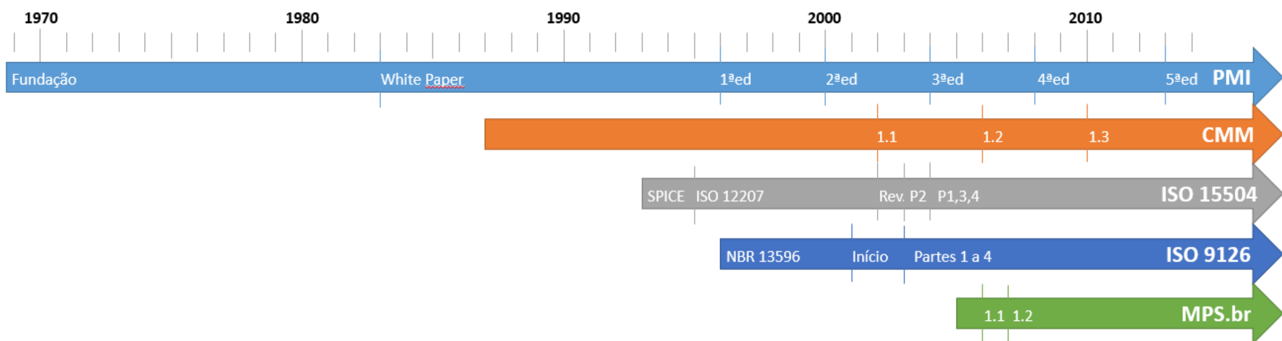


Imagem 2 - Comparação do quadro histórico

Com a divisão em 4 partes da ISO/IEC 9126 em 2003, [2] ficou mais fácil de mapear os principais aspectos de qualidade de um software. A primeira parte desta norma cancela e substitui a NBR 13596 - Avaliação de produto de software, mas sem o processo de avaliação (especificado na NBR ISO/IEC 14598 que foi cancelada em 2013 [3]), focando em 6 área de características de qualidade do software e suas subdivisões:

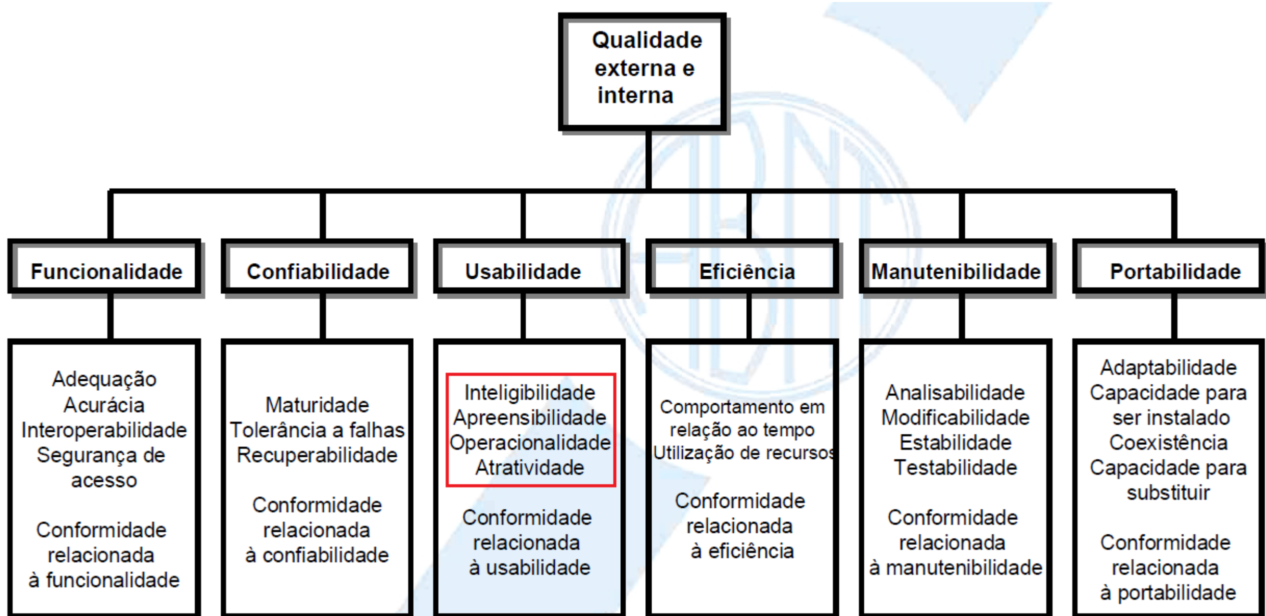


Imagem 1 - Modelo de qualidade para qualidade externa e interna

Analisando a figura 1, pode-se notar que de alguma forma o programador precisou estar envolvido em todas as áreas, mesmo que parcialmente como nos 4 itens de usabilidade, que podem ter sido definidos por outro profissional, mas que foi implementado pelo programador.

O programador está ligado diretamente ou indiretamente a maioria dos itens que tratam a qualidade interna e externa, da lógica do código fonte aos componentes visuais da interface do

usuário, passando por todos os requisitos de infraestrutura e suas características física e logica, do banco de dados, rede, outros hardwares envolvidos, as API, SDK, Frameworks, SO, IDEs, e a linguagem, irão interferir na qualidade final do software, portanto por mais que haja um bom analista que faça um ótimo levantamento de requisitos, é de responsabilidade do programador validar a análise para que o desenvolvimento posso ocorrer com a qualidade real esperada.

A ISO/IEC 12207 [6], divide o ciclo de vida de software em 5 partes, e explora cada uma destas partes:

1) Processo de aquisição
2) Processo de fornecimento
3) Processo de desenvolvimento
4) Processo de operação
5) Processo de manutenção

O PMBok está na 5ªedição [4], o Project Management Institute (PMI) foi fundado em 1969 e tem sido a maior referência sobre processos de qualidade para qualquer área. [5]

A ISO/IEC 15504 [7], é a evolução da ISO/IEC 12207 criando níveis de maturidade, por meio de muita documentação, medição, verificação, validações para garantir a qualidade.

Com o CMMI (Capability Maturity Model Integration) [8], entende por capacidade de um processo a habilidade com que este alcança o resultado desejado, definido assim:

- Capacidade é o "quanto" de alguma coisa (é um valor quantitativo)
- Habilidade é "poder fazer" alguma coisa (é o saber)
- Maturidade é a melhor Capacidade de realizar as Habilidade

Portanto literalmente o "Modelo Integrado Maduro de Capacidade", irá reunir todas habilidades (conhecimento técnico), necessários em um projeto de software, a fim de criar um software de qualidade desejado, por meio de processos bem definidos que deverão ser cumpridos por todos, de estagiários aos profissionais mais experientes, para fazer um software de qualidade de acordo com níveis de maturidade, onde cada nível é um conjunto de processos que estão sendo seguidos.

O MPS.br [9] é uma adaptação para a realidade brasileira do CMMI de forma mais acessível a empresas conseguirem certificar-se como desenvolvedoras de software com qualidade de acordo com nível de detalhamento ou quantidade de processo que a empresa segue.

A tecnologia evoluiu muito rápido, surgem novas linguagens, novos sistemas operacionais, outros são atualizados frequentemente, novos hardwares, e é preciso não apenas criar novos projetos, mas garantir que estes no mínimo acompanhem a tecnologia por muitos, mas nem sempre a equipe que projetou será a mesma que irá dar continuidade. Poucas pessoas ficam décadas no mesmo emprego, mas os processos de qualidade também ajudam a organizar os projetos e a empresa de forma que qualquer pessoa pode ser substituída, pois tudo deve estar documentado de alguma forma, para que a próxima equipe tenha condições de entender e continuar o projeto.

Assim de forma geral os processos de qualidade deixam claro que quando não há um processo não é possível garantir, medir, comparar e qualificar de forma quantitativa nada: nem qualidade, nem entrega, nem custo, nem funcionamento, nem segurança, nem usabilidade, nem performance, nem continuidade como mostrado nos quadros a seguir:

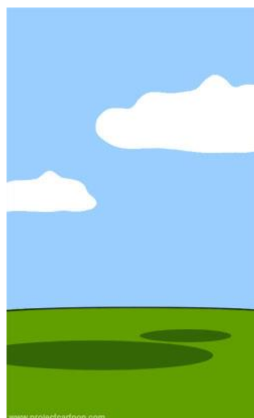
O que o cliente explicou

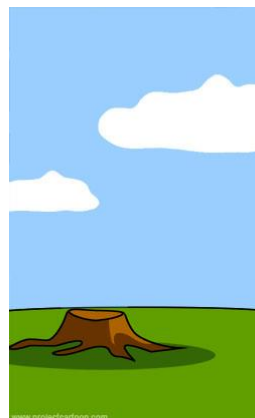
Como o analista planejou

O que o programador codificou

Como o consultor de negócio descreveu

O que foi instalado


Como foi documentado


Como foi mantido

O que o cliente precisava

Imagem 3 – Sátira “The Project Cartoon” [10]

Um documento da NASA [11] sugere vários padrões a ser adotados por seus programadores, e em todas as práticas é interessante notar o uso de 2 conceitos fundamentais:

- “Shall” (deve) – Qualquer software primeiro DEVE cumprir os requisitos, a finalidade Principal que é a necessidade do negócio
- “Should” (recomenda-se) – Após ter cumprido o requisito principal, pode-se e inventar alguma coisa a mais que em algum momento alguém até poderia recomendar como um novo requisito.

Ou seja, é permitido inovar, desde que não se perca o foco do negócio, pois o requisito é o que o negócio precisa, qualquer coisa além por mais útil que possa ser não deve ser prioridade, por mais que possa agregar valor, segurança ou performance, uma inovação que gere risco ao negócio apenas por ser mais ‘legal’ pode matar o projeto.

Na engenharia de software existe conceito de “Ponto de Função” (APF), que tenta avaliar de forma quantitativa cada funcionalidade que o sistema deve ter por mais simples que seja, para que o foco nunca seja no que o sistema deveria fazer em vez do que deve ser feito.

III. O PROGRAMADOR, HERÓI OU VILÃO?

Há muitos programadores, e estes com muitas habilidades, mas que nem sempre seguem algum processo. Então como garantir todo o ciclo de vida do software, desde o levantamento de requisitos, definição o escopo, administração dos custos, desenvolvimento, testes, monitoramento, validação, gestão das alterações do escopo e cronograma, para enfim poder entregar o que foi projetado?

Alguém com muita experiência em uma determinada tecnologia nem sempre irá dar a solução mais simples ou com o melhor custo para um problema simples, será "como matar uma mosca com um canhão", e da mesma forma um profissional sem experiência não conseguirá especificar corretamente um projeto de grande porte.

Programar rápido pode parecer uma boa característica de um programador, e até parece uma habilidade sobrenatural de fluência em uma linguagem, as vezes programadores com alta produtividade ou com muita criatividade ou conhecimento são como um super-herói, mas na verdade isso pode ser uma grande armadilha, pois quanto mais código é desenvolvido sem testes, ou quanto mais rápida pode ser a entrega de uma solução, maior a probabilidade de haver erros de casos não esperados.

Excesso de inovação também é um problema comum, usar ferramentas ou linguagens novas, ou que ainda estão na primeira versão é sempre um perigo em projetos que irão ter que coexistir com vários outros sistemas por muitos anos.

Um programador que guarda para si todos os segredos do programa, não documenta nada, cria pensando apenas no prazo, pode estar criando um software que terá problemas de segurança e performance, e assim um dia todo o programa irá parar e não terá como continuado, e o chamado herói se torna um vilão.

Quando programadores se juntam para realizar projetos e tentam usar modelos ágeis (SCRUM, XP), o fazem de forma errada para iludindo-se que há um projeto completo, quando na verdade tudo está solto e desconexo e com certeza juntar tudo no final irá exigir muita codificação adicional e conhecimento extra, onde o mais heroico sempre terá que resolver as pendencias, pois pensam que são os únicos que podem resolver tudo, e é pior quando isso é verdade.

Trabalhar em equipe é desafiante para os heróis pois inconscientemente eles se acham donos da verdade, do código perfeito, da lógica inquestionável, mas é necessário ser humilde para trabalhar em equipe, aceitar críticas e estar aberto seja para opiniões tradicionais como de inovação.

Seja em grupo ou sozinho, muitos usam o processo de desenvolvimento em espiral, também de forma errada como quem diz: "vai fazendo, no próximo ciclo a gente pensa melhor", e com isso vão surgindo remendos a cada ciclo, e quando se chega ao final do projeto, o código está tão remendado que dificulta o suporte, e inviabilizando novas implementações, e qualquer ajuste se torna apenas um remendo, que afetando a performance, reuso, consistência, segurança, usabilidade.

Quantas vezes protótipos e projetos antigos parecidos são transformados em novos projetos reutilizando códigos inteiros ou parcial por meio de adaptações em vez de se criar uma estrutura de componentes ou framework.

Hoje em dia há muito código grátis ou até de baixo custo, pronto para uso de forma integral ou em partes (componentes), que permite uma rápida entrega, mas nem tudo realmente é bom, cabe ao programador saber avaliar quando é melhor adaptar ou criar do zero alguma solução. Entender as dependências e as limitações de um código faz parte da análise de risco.

E ainda há sempre o risco da saída deste programador herói do projeto, seja ele contratado para outro projeto, ou afastado por doença, ou falecimento, o problema é que uma pessoa não pode ser tão fundamental ao ponto de deter todas as informações de forma que o projeto não tem como continuar na falta desta pessoa. Que o presente Herói, não seja lembrado no futuro como um Vilão.

IV. CONCEITO DO HÁBITO TRIANGULAR

Habito são rotinas que fazemos praticamente de forma automática, quase sem pensar.

A maioria dos hábitos são chamados de “Hábito Triangular” [12] pois em geral são rotinas (coisas) que fazemos quando algum gatilho é acionado para que possamos receber uma recompensa.

Processo de Criação de um Hábito

Gatilho ➔ Rotina ➔ Recompensa

Com essa informação entende-se que é possível criar novos hábitos e até modificar existentes, como objetivo principal deste artigo é a qualidade do software a ideia é criar hábitos de qualidade (rotinas) para que no mínimo o programador tenha um a boa reputação pessoal (recompensa) por fazer um software de qualidade, embora as vezes uma recompensa mais palpável possa ser outros benefícios como por exemplo dinheiro, produtividade, ou maior facilidade de identificar problemas futuros, e qualquer outra vantagem de ter um código de melhor qualidade.

Assim o gatilho pode ser a lembrança do que é um código ruim, mal feito, preguiçoso, sem segurança, ou outras situações que cada um já passou, e que aprendeu a lição do que não deve mais ser feito, agora é refletir nas vantagens de fazer uma nova “Rotina”, de criar um novo hábito, visando receber as recompensas.

V. CRIANDO HÁBITOS DE QUALIDADE

Conhecendo assim um pouco dos padrões e processos de qualidade, além problemas comuns encontrados em projetos de software sem processos, pode-se propor a criação e modificação de hábito em programadores, identificando gatilhos, para que seja criado uma nova rotina cotidiana, independentes do projeto ou de qualquer outro processo de qualidade, a fim de se ter como principal recompensa a garantia da qualidade.

A fim de exemplificar de forma prática como criar estes hábitos, a seguir é mostrado uma lista hábitos com seus gatilhos, rotinas, e recompensas.

Hábito ao definir os requisitos

Em quase todo projeto os requisitos são definidos errados, ou mudam durante o projeto. O cliente que pensa em apenas ter um “Sistema de Pedidos” não consegue enxergar que há muita coisa embutida nisto e que o programador terá que implementar o Layout da Interface de usuário,

cadastro de usuários e nível de acesso, segurança, cadastros auxiliares de produtos, subprodutos, clientes, tabelas de preços, propostas, vendedores, comissões, relatórios, gráficos, cobrança, pagamento, funcionamento off-line, mobile, sincronismo, acompanhamento do cliente, etc.

Além de tudo faz parte dos requisitos a definição completa do contexto e da linguagem, por exemplo, não dá para fazer um sistema multi-plataforma em .Net, por outro desenvolver em C/C++ geralmente não é viável, mas interface com hardware em Java também tem suas limitações se a plataforma não oferecer suporte nativo, linguagens de script não compiladas como PHP ou ASP não são recomendadas para sistemas muito grande e que exigem muita performance, o programador precisa ter um 'check-list' de perguntas a fim de definir bem o contexto e todos os requisitos e abrangência do software para poder estimar cada ponto de função de forma real.

Gatilho:

- Definição dos requisitos de um projeto
- Definições da plataforma, contexto, linguagem
- Definição de Prazo e Custo
- Definição da Infra-estura, performance, carga

Rotina:

- Pensar em cada ponto de função externo ou interno do software, tudo leva tempo para fazer
- Criar lista de perguntas a fazer nas reuniões de requisito, não pode-se esquecer de nada
- Focar em definir requisitos do negócio, melhorias e inovações não são o foco

Recompensa:

- Não ter surpresa, stress ou conflito na hora de entregar o programa ou de cobrar

Hábito ao definir os riscos

Qualquer projeto envolve riscos, por mais simples que possa ser, algo pode dar errado, o importante é definir os limites destes riscos e os meios de contingência quando necessários

Gatilho:

- Definição dos Requisitos
- Definição dos Prazos, Custos e SLA

Rotina:

- Registrar quaisquer riscos que poderiam ser sanados se outros requisitos fossem aprovados
- Preparar e testar rotinas de contingência: Backup/Restauração

Recompensa:

- Tranquilidade prevista, caso surja qualquer problema todos estavam cientes.

Hábito de comentar o código

Em ambiente onde não há documentação, qualquer e-mail, rabisco ou anotação é importante, mas com o tempo estas informações se perdem, ao passar por várias equipes ou programadores, por outro lado os comentários que o programador coloca dentro de cada código estarão sempre lá, próximo do programador que dará suporte e continuará o projeto.

Antes de escrever o código, imagina-se a logica, então porque já não deixar registrado estes pensamentos, antes de codificar, há dias que há mais inspiração, dias mais produtivos onde a criatividade de propor uma solução, desenvolver a lógica necessária é melhor, mas não é qualquer um que entenderia, assim comentários explicando o raciocínio lógico vão ajudar, principalmente se no futuro quem for dar manutenção seja outra pessoa.

Algumas IDE até geram arquivos de documentação baseado em comentários estruturados, como por exemplo o JavaDoc para Java, Sandcastle para .Net

Ter em mãos milhares de linhas de texto feito por algum desconhecido e ter que confiar que isto está certo, ou que parcialmente funciona, ou já funcionou em algum contexto, e que só precisa de um ajuste, é difícil entender como funciona um sistema quanto toda a documentação não existe mais e nem há os comentários junto ao código.

Gatilhos:

- Não dá tempo para documentar
- Faça rápido sem documentação
- O cliente não quer pagar por uma documentação;

Rotinas:

- Comente o código, pois isso pode ser tudo que ficará registrado para suporte futuro
- Adicione links de referências online.
- Registre em forma de comentário no código principal, todas as ferramentas necessárias para a edição, compilação, testes, versões da IDE e SDK, e quaisquer outros softwares necessários para a configurações do contexto de desenvolvimento e produção.

Recompensa:

- Ter um programa que pode ser continuado

Hábito de criar o mínimo de UML é melhor que nada

Criar arquivos textos junto ao código, ou diagramas de caso de uso, diagrama classes, diagramas de sequência, ou qualquer arquivo auxiliar UML e manter isso junto ao código é fundamental, não é necessário detalhar tudo com RUP, mas ter princípios do sistema bem diagramados é sempre bom para avaliar a analise atual e o entendimento futuro.

Quando o código fonte vai passando de mão em mão, qualquer coisa extra que não estiver junto aos códigos fontes acaba se perdendo, principalmente se estes arquivos de apoio forem grandes dificultando o backup e envio, ou tiverem sido feitos por ferramentas não convencionais ou ferramentas proprietárias, ou de alto custo que outras equipes não consigam abrir facilmente, por

isso criar diagramas até em formato texto, ou algum formato vetorial aberto, desde que seja editável, e mantido junto ao código para que também possa ser facilmente editado durante uma atualização.

Algumas IDE já possuem editores de diagramas que transformam classes em diagramas UML e vice versa, e alguns até mantem automaticamente estes diagramas atualizados.

Gatilho:

- Ao criar novos projetos, novas classes, novas funcionalidades

Rotina:

- Buscar diagramas e definições junto ao código para melhor entendimento do funcionamento dos requisitos
- Criar ou Atualizar algum UML com as novas implementações

Recompensa:

- Ser lembrado como alguém organizado, claro, e eficiente
- Ter um software continuado por outros programadores.

Hábito de programar orientada a aspecto

Cada rotina deve se limitar a resolver apenas seu objetivo principal, assim qualquer item secundário deve ser resolvido por outras rotinas com poucas linhas e de fácil entendimento, e fica melhor ainda se estas forem comentadas.

Ter rotinas com muitos parâmetros e que fazem muita coisa com certeza são rotinas complexas de entender e que podem sim dar problema

Ao refratorar uma rotina longa, surge a possibilidade de fazer rotinas automatizadas de testes unitários, o que garantem que novas funcionalidades não vão alterar os resultados das atuais.

Gatilho:

- Quando estiver dando muito scroll para entender o código
- Quando estiver criando uma ideia de laços complexos com muitos loops e condições

Rotina:

- Mantenha o foco, tenha clareza, não faça tudo em uma única rotina de difícil entendimento.
- Evite muitos laços encadeados
- Refatorar sempre que possível
- Tenha rotinas curtas, que caibam em duas telas de texto
- Faça rotinas de testes unitários criando código com baixo acoplamento (dependências)

Recompensa:

- Fácil entendimento do código reduzindo o tempo de suporte futuro

- Possibilidade de criação de Testes automatizados

Hábito de fazer testes automatizados

Nem sempre é possível desenvolver tudo orientado a teste (TDD - Test Driven Development), pois tal técnica no início parece ser menos produtiva já que envolve projetos em paralelos e modularizados para usem rotinas em comum.

Projeto de Teste ➔ [Modulo a ser Testado] ← Projeto Final

Se não dá tempo de fazer testes em tudo, pelo menos fazer nas rotinas críticas e complexas, e que ainda podem crescer com novas implementações, vale sim investir tempo e criar testes unitários que irão poupar futuramente muito tempo para garantir que uma nova funcionalidade não corrompa o funcionamento do que já estava funcionando.

Algumas IDE ou Suítes de Testes contem recurso de “code coverage”, (Visual Studio) que disponibilizam relatórios que apontam quais rotinas, condições e laços foram usados durante o teste, pontuando assim o quanto foi a abrangência do teste, se conseguiu alcançar o máximo das funcionalidades especificadas.

Gatilho:

- Criação de rotinas complexas
- Criação de rotinas que provavelmente irão ter novas funcionalidades
- Ao refatorar ou refazer uma rotina

Rotinas:

- Criar rotinas com baixo acoplamento para facilitar a criação de testes unitários.
- Desenvolver e depurar usando testes unitários em vez de ficar simulando dados via interface ou banco.
- Usar recursos de “Assert” disponível em várias linguagens

Recompensa:

- Maior produtividade e precisão para encontrar erros

Hábito de ter um ambiente produtivo

Por menor que seja o ambiente de trabalho, os computadores, servidores, e as ferramentas envolvidas, precisam colaborar para que o trabalho seja produtivo.

Por exemplo ter ou monitor grande, ou dois monitores, permite visualizar a tela de código ao mesmo tempo que se vê a tela de teste ou depuração do programa, facilitando a edição do código, aumentando assim a produtividade.

Aprender teclas de atalhos, aumenta muito a produtividade geral, pois não é segredo que é muito mais rápido teclar do que mover o mouse até uma opção e clicar, isso quando não tem que

dar vários cliques, em diversas janelas para fazer uma ação que uma única sequência de tecla de atalho já não faz direto.

A performance do computador de desenvolvimento é importante, pois esperar o computador trabalhar pode desmotivar, e hoje em dia o custo de um desktop ou notebook com uma boa performance não é tão discrepante de uma máquina básica.

Experimente outras IDE e ferramentas, aprenda mais profundamente os recursos que e as funcionalidades das ferramentas de desenvolvimento, decore e use teclas de atalho, tente abandonar o mouse, esteja atendo a novas opções que talvez lhe traga uma maior produtividade.

Fazer duas tarefas ou mais ao mesmo tempo é até mais demorado para um computador devido a troca de contexto, quanto mais para um ser humano que não consegue prestar total atenção em dois assuntos simultâneos, foque em fazer uma coisa de cada vez na sequência e o tempo total gasto será menor do que fazer os dois ao mesmo tempo.

Gatilho:

- Quando tiver um tempo livre
- Quando você não estiver sendo produtivo

Rotina:

- Aprender mais sobre a IDE e ferramentas de desenvolvimento
- Pesquisar, assistir vídeos e tutoriais para se aprofundar no conhecimento das linguagens
- Testar e aprender novas funcionalidades, teclas de atalho, instruções, parâmetros
- Evite distrações ou fazer mais de uma coisa ao mesmo tempo, mantenha o foco

Recompensa:

- Estar sempre atualizado
- Possuir mais domínio e produtividade com as ferramentas e linguagens
- Entregar mais rápido

Hábito de ter uma boa reputação

Quando um programador começa a trabalhar no código de um outro programador há somente duas possíveis impressões sobre o projeto desenvolvido: “boa” ou “má”.

É muito fácil condenar um código feito por qualquer outro programador, mas quando se trata do seu próprio código? Qual impressão um outro programador teria ao abrir seu código?

Se tivermos bons hábitos ao programar, nosso código será bem legível a qualquer programador, e será uma referência de organização e clareza de ideias e estruturação.

Gatilho:

- Sempre que estiver programando

Rotina:

- Se esforce para fazer um código de qualidade comentando, refatorando e mantendo tudo organizado para fácil manutenção futura

Recompensa:

- Seja lembrado como um programador de qualidade, um verdadeiro herói!

VII. CONCLUSÃO

Programadores que construírem hábitos de qualidade irão produzir softwares com mais qualidade, e estes poderão ser continuados por outros programadores, e para isso é necessário identificar e conhecer seus próprios hábitos ao codificar, criar novas rotinas de processos mental que lhe recompense, de forma que a qualidade esteja dentro do programador como um valor enraizado e embasado em conhecimento solido sobre o assunto, pois da mesma forma que estes valores se transformaram em processos, é também possível transformá-los em hábitos pessoais para o programador aplicar em qualquer projeto evoluindo como pessoa e profissional de qualidade que facilmente se adaptará a empresas que já seguem normas e processos de qualidade.

REFERÊNCIAS

[1] Dicionário online de português, <http://www.dicio.com.br/habito>. Acesso em 24 Mai 2014

[2] ABNT – Associação Brasileira de Normas Técnicas, NBR ISO/IEC 9126-1, Engenharia de software - Qualidade de produto. Parte 1: Modelo de qualidade

[3] ABNT Catálogo, ABNT NBR ISO/IEC 14598-3:2003. Disponível em <http://www.abntcatalogo.com.br/norma.aspx?ID=002785>. Acessado em 7 Jun 2014

[4] PMI Capítulo São Paulo, 5ª edição do Guia PMBOK®. Disponível em <http://www.pmisp.org.br/slideshow/1617-5-edicao-do-guia-pmbok-em-portugues-esta-disponivel-para-download>. Acessado em 7 jun 2014

[5] Um Guia do Conjunto de Conhecimentos em Gerenciamento de Projetos, Terceira edição (Guia PMBOK), Project Management Institute, Inc, 2004

[6] ABNT – Associação Brasileira de Normas Técnicas, ISO/IEC 12207 - Tecnologia de informação – Processos de ciclo de vida de software

[7] Wikipédia, ISSO/IEC 15504. Disponível em <http://pt.wikipedia.org/wiki/ISO_15504>. Acessado em 26 Abr. 2014

[8] Wikipédia, CMMI. Disponível em <http://pt.wikipedia.org/wiki/CMMI>. Acessado em 7 jun 2014

[9] [SOFTEX, 2009] - ASSOCIAÇÃO PARA PROMOÇÃO DA EXCELÊNCIA DO SOFTWARE BRASILEIRO – SOFTEX. MPS.BR – Guia Geral, maio 2009. Disponível em:<http://www.4shared.com/office/qObANSnb/MPSBR_Guia_Geral_2009.html?locale=pt-BR>. Acesso em: 26 Abr. 2014.

[10] Donaldo M. Dagnone, The Project Cartoon, <http://www.projectcartoon.com>. Acesso em: 26 Abr 2014

[11] NASA, JPL Institutional Coding Standard for the C Programming Language, Version 1.0 March 3, 2009. Disponível em <http://lars-lab.jpl.nasa.gov/JPL_Coding_Standard_C.pdf>. Acessado em 7 jun 2014

[12] Arata Academy, resumo, Charles Duhigg, Poder do Hábito, Editora Objetiva. Disponível em <http://www.produtividadeninja.com/baixar-comocriaremodificarhabitos>. Acesso em 26 Abr 2014