Programação ágil: A evolução histórica

Ana Paula Afonso [1]

apafonso@iscap.ipp.pt

[1] Instituto Superior de Contabilidade e Administração do Porto, Porto, Portugal

Resumo: A Programação Ágil – Extreme Programming (XP) – é uma das representantes dos actuais processos ágeis de desenvolvimento de software, que visam minimizar os riscos encontrados na actividade da programação de sistemas. O XP não nasceu no meio académico e, foi criada por Kent Beck em 1997 e embora tenham vindo a decorrer, há alguns anos, conferências dedicadas ao assunto, o número de publicações ainda é reduzido quando comparado ao de outras áreas da Informática. Na indústria, os resultados de sua adopção têm sido animadores, mas a comunidade científica tem demonstrado um posicionamento céptico visto que diversas práticas propostas contrariam conceitos amplamente difundidos e utilizados tanto nas universidades, quanto na indústria. Este artigo tem como propósito principal apresentar, genericamente, a metodologia XP, a sua origem e respectiva evolução histórica até aos nosso dias, com base na forma como as soluções para os projectos de software têm vindo a ser organizadas, assim como na análise dos problemas recorrentes na maioria dos projectos de software.

Palavras- chave: Engenharia de software; *Extreme Programming* (XP); Programação ágil; *Lean* software; *Rational Unified Process* (RUP)

1. Introdução

Desenvolver software é uma actividade complexa e arriscada. De uma forma geral os maiores riscos são geralmente gastos que ultrapassam o orçamento, consumo de tempo que ultrapassa o cronograma, funcionalidades que não resolvem os problemas dos utilizadores e a baixa qualidade apresentada pelos sistemas desenvolvidos.

A indústria de software tem vindo a pesquisar, há algumas décadas, técnicas de desenvolvimento que reduzam tais riscos, tornando assim essa actividade mais produtiva. Nesse sentido, a criação da disciplina de Engenharia de Software em 1968 é um marco de referência. A partir daí, foram apresentadas inúmeras propostas começando pelo processo de desenvolvimento linear e sequencial (em cascata) até chegar aos actuais processos ágeis de desenvolvimento.

A Programação Ágil – Extreme Programming (XP) – é uma das representantes destes processos e foi criada por Kent Beck em 1997 num projecto para a Chrysler (fabricante de veículos norte americana). O XP é constituído por uma conjunto reduzido de práticas de desenvolvimento organizadas em torno de quatro valores básicos – feedback, comunicação, simplicidade e coragem. Estas práticas fortemente inter-relacionadas formam um grupo de elevada sinergia.

Este artigo tem como propósito principal apresentar, genericamente, a metodologia XP, a sua origem e respectiva evolução histórica até aos nosso dias, com base na forma como as soluções para os projectos de software têm vindo a ser organizadas, assim como na análise dos problemas recorrentes na maioria dos projectos de software.

2. A crise do software

O termo "crise do software" tem vindo a ser utilizado na indústria de software desde 1968, quando se admitiu abertamente que existia uma crise latente na área (Dijkstra, 1972). Naquele ano, ocorreu a Conferência da NATO sobre Engenharia de Software (*NATO Software Engineering Conference*) em Garmisch, Alemanha, que é considerado actualmente o momento histórico do nascimento da disciplina de Engenharia de Software (Brian, 1996).

O Standish Group, uma empresa localizada em Massachusetts, EUA, publica desde 1994 um estudo chamado de *CHAOS Report*. Trata-se de um amplo levantamento envolvendo milhares de projectos na área de tecnologia da informação. Actualmente, os seus dados estão entre os mais utilizados para quantificar a "crise do software". A última informação foi de que a taxa de sucesso obtida em projectos em TIC é de cerca de 30%, os gestores de projecto à volta do mundo sentem-se na obrigação de conseguir controlo sobre esta situação através de processos de gestão num âmbito concreto (Dekkers, 2007).

O *CHAOS Report* do ano 2006 citado em (Goldsmith, 2007) apresenta uma percentagem de sucesso à volta dos 34% revelando ainda que:

- Os atrasos representam, em média, 63% mais tempo do que o estimado;
- Os projectos que não cumprem o orçamento custaram em média mais 45%;
- Em geral, apenas 67% das funcionalidades prometidas são efectivadas.

O Standish Group classifica o resultado final de um projecto como sendo Mal sucedido, Comprometido ou Bem sucedido. Em 2006, o resultado final da pesquisa mostrou a seguinte distribuição entre os projectos (Figura1). Tais números, embora desastrosos, mostram um avanço em relação aos resultados do primeiro levantamento realizado em 1994 (Figura 2).

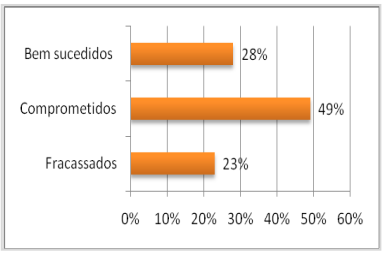

Figura 1 - Estatística sobre o resultado final de projectos de software

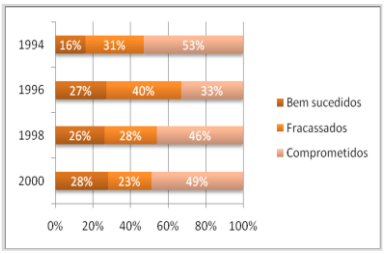

Figura 2 - Evolução do resultado final de projectos de software entre 1994 e 2006.

ser chamado há quase quarenta anos de "crise do software", a análise de casos isolados demonstra a existência de grandes variações nos resultados dos projectos.

Ao analisar os factores que levam tantos projectos de software a fracassarem e outros (uma minoria) a serem bem sucedidos, leva-nos a repensar o que significa desenvolver software. Quais são os factores que caracterizam o desenvolvimento de software e que diferenciam os projectos desta área?

3. A natureza do software

A compreensão dos factores que levam à crise do software passa, em primeiro lugar, pela compreensão da natureza do software e como tem vindo a ser tratado ao longo do tempo. Esta análise envolve duas partes, os aspectos básicos que caracterizam o software e as metáforas que são utilizadas no desenvolvimento do mesmo.

A partir da análise do software em estudos anteriores (Brooks, 1987; Krutchten, 2001), o estudo detalhado do software aponta para as seguintes características como sendo responsáveis pelo insucesso no desenvolvimento de sistemas:

- *Grande complexidade;*
- *Ausência de conformidade;*
- *Invisibilidade;*
- *Ilusão de maleabilidade.*

A partir da análise desses quatro pontos, discutiremos em seguida as dificuldades associadas ao desenvolvimento de software:

Grande Complexidade – De entre as características acima referidas, Brooks (1987) destaca a questão da complexidade por ser decisiva no processo de desenvolvimento de software. Ele considera que os sistemas de software

normalmente possuem uma quantidade elevada de elementos distintos, o que os torna mais complexos que qualquer outro tipo de construção humana.

Quando as partes de um software são semelhantes, normalmente são agrupadas em métodos, classes e outros elementos. Assim, à medida que um sistema aumenta em tamanho, aumenta também a quantidade de partes distintas. Esse comportamento difere profundamente de outras construções, tais como computadores, prédios ou automóveis, nos quais se encontram elementos repetidos em abundância. O resultado directo de tal situação é o número extremamente elevado de estados que um programa apresenta.

Isso gera uma grande complexidade, que se torna maior à medida que o número desses elementos cresce, obrigando a uma maior interacção entre os elementos.

Sendo assim, cada nova funcionalidade acrescentada ao software exige, além do esforço na sua construção, à necessidade de integração com as existentes. A complexidade resultante pode estender-se a outros níveis se considerarmos o elevado número de possíveis estados que o software pode alcançar.

As equipas de desenvolvimento de software têm grandes dificuldades em lidar com grandes sistemas. Ao contrário do que ocorre numa linha de montagem, o simples acréscimo de mais pessoas a uma equipa não torna o trabalho mais fácil.

Ausência de conformidade – A complexidade é um problema que afecta diversas áreas do conhecimento científico, entretanto a maioria delas é baseada na convicção de que existem princípios unificadores que, uma vez descobertos, facilitam a sua compreensão. Inversamente, os sistemas de software não costumam existir em conformidade com princípios básicos e imutáveis. Sendo impossível formar uma base sólida de conhecimento como a encontrada na engenharia tradicional. Essa inexistência de princípios básicos faz com os padrões de software sejam baseados unicamente em boas práticas.

“Grande parte da complexidade existente no software é arbitrária. Ela é imposta por instituições e sistemas humanos. Tal conformidade é dinâmica, visto que os sistemas humanos mudam com o tempo, as pessoas mudam e o software passa a ter que se adequar a novas realidades” (Brooks, 1987).

Como se não bastasse, a dinâmica dos sistemas representados pelo software e a rápida evolução tecnológica gera mais uma incerteza: as próprias técnicas de desenvolvimento e ferramentas mudam em ritmo acelerado. Isso gera uma enorme pressão sobre as empresas e equipas de desenvolvimento, que são “obrigadas” a receber formação, constantemente, para se adequarem às tecnologias mais recentes.

À medida que uma equipa de desenvolvimento se actualiza, as empresas enfrentam outro problema: lidar com os sistemas herdados. As tecnologias usadas em larga escala há dez ou vinte anos atrás são hoje obsoletas, e existe uma enorme

dificuldade em encontrar profissionais que as consigam manter em sistemas ainda em uso. O bug do ano 2000 é já um exemplo clássico.

I**nvisibilidade** – Ao contrário de outros produtos o software não pode ser desenhado ou projectado de uma maneira que corresponda a mundo real, ao não possuir natureza física e não pode ser definido geometricamente. Mesmo os diagramas criados para representação, na verdade não descrevem o software em si, mas fluxos de controlo, fluxos de dados ou padrões de dependência. Isso dificulta o trabalho das equipas de desenvolvimento de software que deve ser baseado maioritariamente em representações abstractas.

A comunicação entre os membros da equipa é prejudicada, dada a dificuldade em falar sobre um elemento abstracto. Originando a que parte do conhecimento sobre o projecto só fique retido na mente de membros específicos da equipa, uma vez que tal conhecimento não é representado de forma satisfatória e clara.

Ilusão de maleabilidade – O software possui uma maleabilidade extremamente elevada. Por não ter natureza física, as pessoas encaram-no como algo de fácil adaptação. Em geral, o indivíduos acham-no extremamente maleável e crêem que qualquer mudança terá um baixo custo. É bastante comum o cliente mudar o âmbito do projecto durante o desenvolvimento, sem considerar as implicações de tal mudança.

> Krutchten (2001) afirma, a propósito que "Se você constrói uma ponte, você não tem este tipo de flexibilidade. Você não pode dizer: "Hum, agora que eu já vejo os pilares, eu gostaria que essa ponte fosse colocada duas milhas rio acima ".

Na situação acima, qualquer um teria a clara percepção do esforço monumental de mover uma ponte de um lugar a outro. Entretanto, no caso do software, a sua invisibilidade como produto cria a ilusão de que é extremamente fácil realizar uma alteração, levando os clientes a solicitá-las com mais freqüência.

Concluindo, é interessante referir que as características acima resumidas têm aspectos que extrapolam os seus próprios limites, fazendo com que exista uma interdependência entre si. Assim, podemos dizer que a ausência de conformidade contribui para a grande complexidade, ou que a ilusão de maleabilidade é conseqüência da invisibilidade.

Possivelmente a invisibilidade é o problema central quando se fala de desenvolvimento de software. A ausência de uma natureza física faz do software um produto intangível, o que por si é um motivo para se tentar encontrar formas compreender o seu funcionamento. As metáforas foram a principal ferramenta para se chegar a essa compreensão. Comparar o software a um outro elemento de mais fácil visualização ajuda a compreendê-lo melhor, e como conseqüência, a construí-lo de forma melhor. Daí o termo "engenharia de software".

4. O desenvolvimento de software

A procura de soluções que possam tornar os projectos de desenvolvimento de software mais produtivos, mais previsíveis e com resultados de melhor qualidade, não é recente. Esses objectivos, entre outros, foram alcançados com sucesso pela indústria de produção em massa.

Segundo Hammer (1994) citado em (Kacuta, L. Y. 2006), desde 1776, quando Adam Smith defendeu a divisão do trabalho em 'A riqueza das nações', racionalizar a produção tem vindo a servir como um método provado para elevar a qualidade, reduzir custos e aumentar a eficiência.

Uma questão relevante, que acompanha a indústria de software há longo tempo, é se não seria possível utilizar estes mesmos princípios no desenvolvimento de software e obter os mesmos resultados positivos. Muitos acreditam que é possível mapear estes princípios no desenvolvimento de software e cada vez mais encontramos metáforas que procuram a sua aproximação ao processo de produção de uma fábrica, culminando inclusivamente na actual ideia de "fábrica de software" (*software factory*) tão amplamente divulgada no mercado.

Os elementos das equipas de desenvolvimento de software encontram-se entre os "trabalhadores do conhecimento". Portanto, é fundamental avaliar os factores que podem ser utilizados para aumentar a produtividade, a previsibilidade e a qualidade do "trabalhador do conhecimento", os quais, não são, necessariamente, os mesmos que regem o "trabalho manual". De facto, como veremos adiante, tais factores, embora ainda não sejam tão bem conhecidos, parecem distanciar-se largamente daqueles tradicionalmente usados para os "trabalhadores manuais".

Segundo Drucker (1999) citado em Carvalho (2002), existem seis factores essenciais que determinam a produtividade do trabalhador do conhecimento. São eles:

1. Definir qual é a tarefa a ser feita;
2. Permitir que os próprios trabalhadores se auto-giram. Ou seja, assegurar que eles tenham autonomia e responsabilidade sobre o que produzem;
3. Assegurar que os trabalhadores tenham a oportunidade de inovar;
4. Aprendizado e ensino contínuo;
5. Qualidade é um factor tão ou mais importante que a quantidade produzida;
6. Os trabalhadores do conhecimento precisam ser tratados como "activos" e não como "custo".

4.1 O desenvolvimento de software "magro" – Lean software

Podemos afirmar que o software "magro", ao conter princípios que estão na base dos processos ágeis de desenvolvimento de software, é um precursor do XP. Tendo como características principais o conjunto seguinte constituído por sete princípios básicos (Poppendieck, 2002):

1. Eliminar desperdícios;
2. Amplificar a aprendizagem;
3. Adiar decisões ao máximo;
4. Entregar o mais rapidamente possível;
5. Delegar poder à equipa;
6. Incorporar integridade;
7. Ver o todo.

Estes princípios incorporam os factores citados anteriormente sobre a produtividade do trabalhador do conhecimento.

4.2 Processos de desenvolvimento de software

Como vimos anteriormente o termo "crise do software" acompanha a indústria de software desde 1968, quando ocorreu a conferência da NATO que definiu o nome Engenharia de Software. O termo Engenharia de Software deu origem a uma disciplina dentro da área da ciência de computadores, embora originalmente tivesse sido criado apenas como uma provocação. Conforme os relatos de Peter Naur e Brian Randell o termo "engenharia de software" foi escolhido deliberadamente para ser provocativo, cuja implicação era a necessidade de que a criação de software fosse baseada nos tipos de fundamentos teóricos e disciplinas práticas que são tradicionais em ramos estabelecidos da engenharia. A Engenharia de Software surgiu com o objectivo de atender às necessidades da NATO para o desenvolvimento de grandes sistemas de defesa.

Segundo o IEEE *Standard Computer Dictionary*, a engenharia de software "é a aplicação de uma abordagem sistemática, disciplinada e mensurável para o desenvolvimento, operação e manutenção de software; isto é, a aplicação da engenharia ao software". O objectivo desta abordagem é alcançar na área de software o mesmo nível de previsibilidade, determinismo e acerto presentes noutros ramos da engenharia.

A Engenharia de Software deu origem a diversos processos de desenvolvimento. O mais divulgado (e antigo) é o processo linear e sequencial de desenvolvimento, também conhecido como cascata. Neste processo os projectos de software são organizados em quatro grandes etapas executadas sequencialmente: análise, desenho, codificação e testes (Pressman, 2005). Ainda hoje, este é o modelo de desenvolvimento mais conhecido e amplamente utilizado.

DeMarco (2002) acredita que a focalização na geração de processos, em vez de ajudar, acabou por se tornar uma forma de agravar os problemas da "crise do software". Segundo este investigador, "Depois de quase 20 anos de obsessão por processos (...), o processo que encontro tipicamente em empresas clientes tornou-

se excessivamente baseado em documentações, gerando uma enchente de documentos que se tornou endémica (...)."

Ao longo do tempo, a indústria de software criou alternativas para o processo de desenvolvimento em cascata. Algumas mantiveram parte das características do modelo sequencial, enquanto outras se baseiam em formas completamente diferentes no tratamento dos projectos de software.

Jacobson et al. (1999) citados em Filho (2000) explicam que o *Rational Unified Process* (RUP), por exemplo, traz avanços em relação ao desenvolvimento em cascata, embora mantenha conceitos herdados do taylorismo. É um *framework* de processos muito abrangente que, como tal, pode ser instanciado para atender às necessidades dos mais diversos tipos de projectos de software

O RUP é organizado em torno do conceito de "**melhores práticas**" providenciando um vasto arcabouço de práticas que procuram indicar a melhor forma de se realizar diversos tipos de actividades nos projectos de software.

Genericamente o RUP (Kacuta, 2006) estabelece o desenvolvimento iterativo e incremental como forma de incorporar *feedback* e aprendizagem ao processo de desenvolvimento. No entanto, é comuns as equipas adoptarem o RUP com iterações muito longas ou simplesmente executarem o projecto inteiro em uma única iteração. Nestes casos, o que se observa é a equipa a executar um projecto de acordo com o processo em cascata, mas basicamente usando os processos do RUP para organizar a documentação.

Uma das maiores falhas do processo de desenvolvimento em cascata e de outras propostas de desenvolvimento baseadas em princípios *tayloristas* é a incapacidade de levar em conta o factor humano de forma apropriada. Taylor tornou produtivo o trabalho manual, mas seus princípios não são aplicáveis para o trabalho do conhecimento, em especial o desenvolvimento de software.

Kacuta, (2006) afirma que "um bom software não tem origem em ferramentas CASE, programação visual, prototipagem rápida ou tecnologia de objectos. Um bom software é o resultado de pessoas. Assim como é o caso de maus softwares. (...) já que o software é criado por pessoas e usado por pessoas, uma melhor compreensão das pessoas – como executam o trabalho e como trabalham em conjunto – é a base para um melhor desenvolvimento de software e para criarmos softwares melhores"

Na década de 90, alguns profissionais da indústria de software propuseram novos processos de desenvolvimento que consideravam o "lado humano" nos projectos de software. Em Fevereiro de 2001, 17 profissionais experientes reuniram-se em Utah (EUA) para discutir práticas de desenvolvimento e propostas alternativas visando evitar os processos de desenvolvimento excessivamente baseados em documentação e formalismos. Decidiram, então, organizar as propostas sob um nome comum: **desenvolvimento ágil de software**. Isto foi feito através do

lançamento do Manifesto pelo Desenvolvimento Ágil de Software3 (Beck et al, 2001).

O manifesto estabelece um conjunto de valores que são os adoptados nos projectos ágeis:

- **Indivíduos e interacções** em vez de processos e ferramentas;
- **Software a funcionar** em vez de documentação abrangente;
- **Colaboração com o cliente** em vez de negociação de contratos;
- **Responder a mudanças** em vez de seguir um plano.

A proposta é que, embora exista valor nos itens à direita, os processos ágeis valorizam mais os itens que estão à esquerda. Os principais processos ágeis mais conhecidos actualmente, são os seguintes: Scrum, Dynamic Systems Development Method (DSDM), Crystal Methods, Feature-Driven Development (FDD), Lean Development (LD), Extreme Programming e Adaptative Software Development.

Uma das principais diferenças dos processos ágeis em relação aos seus antecessores é o conceito chamado de *barely sufficient*, ou seja, **mínimo necessário**.

Enquanto abordagens como o RUP (entre outras) procuram estabelecer uma estrutura de "melhores práticas", os processos ágeis sugerem o uso de um conjunto bastante reduzido de práticas. Tal conjunto pode revelar-se suficiente em muitos projectos comerciais que envolvam equipas reduzidas, tais como os que foram mencionados no início desta secção.

O objectivo é iniciar os projectos de software de forma simples, com poucas práticas e avaliar os resultados. No caso de faltarem práticas ou processos ao projecto, devem "herdar-se" de outras propostas metodológicas. Com isso, procura-se evitar a possibilidade de iniciar um projecto com um conjunto desnecessariamente elevado de práticas e processos que possam torná-lo burocrático. A ênfase passa a ser em trazer elementos novos para o projecto, só quando os mesmos se mostram realmente necessários (Highsmith, 2002). Na próxima secção será apresentado sucintamente o *Extreme Programming* que representa um dos processos ágeis mais conhecidos e utilizados.

5. eXtreme Programming (XP)

Os fundamentos principais do XP tiveram origem nas tradições do desenvolvimento em Smalltalk, e datam de meados da década de 80, quando Kent Beck e Ward Cunningham trabalhavam na Tektronixs, Inc. Práticas, tais como, *refactoring*, programação em par (*pair programming),* mudanças rápidas, *feedback*, participação do cliente, desenvolvimento iterativo, testes automatizados, entre outros, são elementos centrais da cultura da comunidade Smalltalk. Deste ponto de vista, o XP pode ser considerado como tendo o modo de agir do Smalltalk generalizado para outros ambientes.

Entre 1986 e 1987, começaram a surgir algumas contribuições fundamentais para aquilo que viria a ser o XP. De 1986 a 1996, Kent e Ward desenvolveram um vasto conjunto de boas práticas (posteriormente condensadas) publicado em 1996, no padrão de linguagem *Episodes*, designado por *Pattern Languages of Program Design 2*.

Ainda neste mesmo período, entre 1989 e 1992, surgiram importantes avanços em *refactoring*. Nesta área, destaca-se a tese de *Bill Opdyke, Refactoring Object-Oriented Frameworks*. Este trabalho demonstrou como pessoas como Kent e Ward obtinham ganhos de produtividade utilizando a *refactoring*. Alguns anos mais tarde, por volta de 1996, Kent publicou o livro *Smalltalk Best Practices Patterns*.

O desenvolvimento orientado a testes é uma técnica que derivou directamente das técnicas de *refactoring*. O primeiro artigo publicado sobre este conceito foi escrito por Kent Beck para a *SmalltalkReport*. Neste artigo, ele introduziu o *framework SmallUnit*. A partir de então, o artigo mais importante sobre o assunto, intitulado "*Test Infected*", descreveu o JavaUnit na Dr. Dobb's.

Ainda nas questões técnicas, a comunidade XP é tradicionalmente conhecida por utilizar fortemente padrões (*patterns*). Este é um outro aspecto importante no surgimento do XP, já que Kent e Ward começaram a aplicar os conceitos de padrões em 1987, quando escreveram um dos primeiros artigos sobre o assunto para a OOPSLA'87 (*Conference on Object-Oriented Programming, Systems, Languages, and Applications*).

Em 1996, todas as partes que foram sendo agregadas ao longo de uma década começaram a fundir-se. No início daquele ano, Kent trabalhava como consultor para problemas de desempenho (*performace problems*) em SmallTalk, quando foi chamado para analisar o desempenho do projecto de conversão da folha de pagamento da Chrysler para SmallTalk. O sistema em questão, conhecido como C3, ou *Chrysler Comprehensive Compensation System* (Sistema de Compensação Abrangente da Chrysler), é conhecido como o berço do XP e foi onde Kent Beck utilizou pela primeira vez, em conjunto, as práticas que actualmente formam a estrutura do XP.

6. Conclusões

A programação ágil surgiu num contexto de simplificação dos processos de desenvolvimento de software. O conjunto de práticas e valores do XP é coeso e possui características que permitem utilizar esta metodologia como uma alternativa válida na procura de maiores taxas de sucesso nos projectos de software.

7. Referencias

Beck, K. (2007).“Extreme Programming explained”. On-line em http://static.scribd.com/docs/oloxgqdcwnp0.pdf e consultado em Maio de 2007.

Beck, K.; Beedle, M.; Bennekum, A. V.; Cockburn, A. Cunningham, W.; Fowler, M.; Grenning, J.; Highsmith, J.; Hunt, A.; Jeffries, R.; Kern, J.; Marick, B.; Martin, R.C.; Mellor, S.; Schwaber, K.; Sutherland, J.; Thomas, D.. “Manifesto for Agile Software Development”. (2001).Disponível em http://agilemanifesto.org/ e consultado em Maio de 2007.

Brooks, F. P. (1987). “No silver bullet: essences and accidents of Software Engineering.” IEEE. Computer, v. 20, n. 4, 1987. p. 10-19. On-line em http://www.virtualschool.edu/mon/SoftwareEngineering/BrooksNoSilverBullet.html e Consultado em Maio de 2007.

Brian, R. (1996).“The 1968/69 NATO Software Engineering Reports”. Dagstuhl-Seminar 9635: "History of Software Engineering" Schloss Dagstuhl, August 26 - 30, 1996. On-line em http://homepages.cs.ncl.ac.uk/brian.randell/NATO/NATOReports/index.html e consultado em Maio de 2007.

Buhrer, K. (2000). “From craft to science: searching for first principles of software

Development”. The Rational Edge. Disponível em http://www-106.ibm.com/developerworks/rational/library/content/RationalEdge/dec00/FromCrafttoScienceDec00.pdf . Consultado em Abril de 2007.

Carvalho, K.(2002). “O Profissional da Informação: O Humano Multifacetado. DataGramaZero - Revista de Ciência da Informação - v.3 n.5 out/02. Disponível em http://www.dgzero.org/out02/Art_03.htm e consultado em Maio de 2007.

Dekkers, C. et al. (2007). Increase ICT Project Success with Concrete Scope Management. ACM - Digital Library On line em http://portal.acm.org/ft_gateway.cfm?id=1302497&type=external&coll=GUIDE&dl=GUIDE&CFID=17990971&&jmp=abstract&coll=Portal&dl=acm&CFID=17990944&CFTOKEN=41923022 e consultado em Fevereiro de 2007.

DeMarco, T.; BOEHM, B. “The agile methods fray”. (2002).IEEE Computer, v. 35, n. 6,2002, p. 90-92. disponível em http://sunset.usc.edu/csse/TECHRPTS/2002/usccse2002-515/usccse2002-515.pdf e consultado em Maio de 2007.

Dijkstra E. W. , (1972). “The humble programmer”. ACM. Communications of the ACM, v. 15, n. 10, 1972. p. 859-866. On-line em http://www.cs.utexas.edu/~EWD/ewd03xx/EWD340.PDF. Consultado em Abril de 2007.

Filho, W.P.P. (2000) “Requirements for an Educational Software Development Process”. On-line em http://homepages.dcc.ufmg.br/~rodolfo/dcc823-1-07/RequirementsForAnEducationalSoftwareDevProcess.pdf e consultado em Maio de 2007.

Goldsmith R. (2007). *REAL CHAOS, Two Wrongs May Make a Right. On line em* http://www.compaid.com/caiinternet/ezine/goldsmith-chaos.pdf e consultado em Fevereiro de 2007.

Kacuta, L. Y. (2006); “Integração de Modelo do Negócio com Especificação de Software: uma proposta para alinhar Sistemas à Estratégia do Negócio”.. Trabalho Final de Mestrado Profissional em Computação. Campinas, S.P.

Khan, A. (2004). “A tale of two methodologies for web development:heavyweight vs Agile”. Disponível em http://www.dis.unimelb.edu.au/staff/sandrine/supervision/PDF/AliKhan_615690FinalReport_2004.pdfe consultado em Maio de 2007.

Krutchten, P.(2001). “The nature of software: what's so special about software engineering.” The Rational Edge, On-line em http://www-106.ibm.com/developerworks/rational/library/4700.html. Consultado em Abril de 2007.

Pressman, Roger S. (2005). Software Engineering: A Practitioner's Approach, 6/e. Roger S Pressman, R.S. Pressman and Associates. ISBN: 0072853182.

Poppendieck,M. (2001). “Principles of Lean Thinking”. On-Line em http://www.poppendieck.com/papers/LeanThinking.pdf e consultado em Maio de 2007.

THE STANDISH GROUP INTERNATIONAL, Inc. “The CHAOS Report (1994)”. The Standish Group International, Inc, 2001. On-line em http://standishgroup.com/sample_research/chaos_1994_1.php consultado em Maio de 2007.