ISBN: 978-65-986235-3-1

Aurum EDITORA

CAPÍTULO 15

METODOLOGIAS ÁGEIS E TRADICIONAIS NO CICLO DE VIDA DO SOFTWARE

AGILE AND TRADITIONAL METHODOLOGIES IN THE SOFTWARE LIFE CYCLE

https://doi.org/10.63330/aurumpub.005-015

Renato Vidal Borges
Engenharia de software

RESUMO

Este trabalho aborda a comparação entre metodologias ágeis e tradicionais aplicadas ao ciclo de vida do software, considerando suas características, aplicações, limitações e contribuições para o desenvolvimento de sistemas computacionais. O objetivo principal foi analisar como essas metodologias influenciam a eficiência, a flexibilidade e a qualidade no processo de produção de software, considerando as demandas contemporâneas de mercado. A pesquisa adotou abordagem qualitativa, de natureza bibliográfica, com base em autores como Pressman (2001), Sommerville (2003), Beck (1999), Royce (1970) e dados do Standish Group (1995). O estudo identificou que as metodologias tradicionais, representadas pelo modelo clássico ou cascata, ainda são úteis em contextos com requisitos bem definidos e necessidade de rigor documental. No entanto, apresentam dificuldades em projetos sujeitos a mudanças frequentes, devido à sua rigidez e estrutura sequencial. Em contrapartida, as metodologias ágeis, como o Extreme Programming e o Scrum, oferecem maior adaptabilidade, promovem a comunicação direta entre equipes e clientes, favorecem entregas rápidas e contínuas e reduzem falhas ao longo do projeto. Os resultados indicam que a escolha metodológica deve considerar o contexto organizacional, o tipo de projeto e o grau de instabilidade dos requisitos. Conclui-se que, embora não haja uma metodologia universalmente superior, as abordagens ágeis têm se mostrado mais eficazes em ambientes dinâmicos, colaborativos e orientados à entrega contínua de valor ao cliente.

Palavras-chave: Engenharia de Software; Metodologias Ágeis; Metodologias tradicionais.

ABSTRACT

This paper compares agile and traditional methodologies applied to the software life cycle, considering their characteristics, applications, limitations and contributions to the development of computer systems. The main objective was to analyze how these methodologies influence efficiency, flexibility and quality in the software production process, considering contemporary market demands. The research adopted a qualitative, bibliographical approach, based on authors such as Pressman (2001), Sommerville (2003), Beck (1999), Royce (1970) and data from the Standish Group (1995). The study found that traditional methodologies, represented by the classic or waterfall model, are still useful in contexts with well-defined requirements and the need for rigorous documentation. However, they present difficulties in projects subject to frequent changes, due to their rigidity and sequential structure. On the other hand, agile methodologies, such as Extreme Programming and Scrum, offer greater adaptability, promote direct communication between teams and clients, favor fast and continuous delivery and reduce failures throughout the project. The results indicate that the methodological choice must take into account the organizational context, the type of project and the degree of instability of the requirements. The conclusion is that, although there is no universally superior methodology, agile approaches have proven to be more effective in dynamic, collaborative environments geared towards the continuous delivery of value to the client.

Keywords: Software Engineering; Agile Methodologies; Traditional Methodologies.

1 INTRODUÇÃO

O desenvolvimento de software é uma atividade complexa que requer organização, planejamento e escolha adequada de métodos que conduzam à entrega de produtos eficazes, dentro dos prazos e custos previstos. Nesse contexto, a Engenharia de Software surge como um campo fundamental, pois oferece princípios, modelos e práticas que orientam a criação de sistemas computacionais confiáveis e de qualidade. Autores como Pressman (2001) e Sommerville (2003) destacam que o software, enquanto produto lógico, precisa ser desenvolvido com base em metodologias bem estruturadas que favoreçam sua manutenção e evolução, ao mesmo tempo que garantam funcionalidade e desempenho.

Diante da diversidade de demandas do mercado e da constante evolução tecnológica, diferentes abordagens de desenvolvimento foram surgindo. Entre elas, destacam-se as metodologias tradicionais — como o modelo em cascata proposto por Royce (1970) —, que se caracterizam por uma sequência rígida de etapas, e as metodologias ágeis — como o Extreme Programming (XP), de Beck (1999), e o Scrum, de Schwaber e Beedle (2002) —, que oferecem flexibilidade, foco no cliente e adaptação rápida a mudanças. O ponto de virada que consolidou as abordagens ágeis foi o Manifesto Ágil, publicado em 2001 por um grupo de especialistas, cujo objetivo era estabelecer valores e princípios mais adequados às necessidades reais de desenvolvimento de software em ambientes dinâmicos (Agile Manifesto, 2004).

Diante disso, este trabalho tem como objetivo principal comparar metodologias ágeis e tradicionais no ciclo de vida do software, analisando suas características, vantagens, limitações e aplicabilidades. Parte-se da hipótese de que, embora as metodologias tradicionais ainda sejam úteis em certos contextos, as metodologias ágeis apresentam maior aderência às necessidades contemporâneas de desenvolvimento, especialmente em projetos que exigem rapidez, interação contínua com o cliente e flexibilidade diante de mudanças.

Justifica-se a relevância deste estudo pelo fato de que a escolha do modelo de desenvolvimento impacta diretamente na qualidade do produto final, no desempenho das equipes e na satisfação do cliente. Além disso, compreender as particularidades dessas metodologias permite às organizações tomarem decisões mais estratégicas e alinhadas aos seus objetivos e realidades operacionais.

A metodologia adotada neste trabalho é de natureza qualitativa, com base em pesquisa bibliográfica, utilizando obras clássicas e contemporâneas da Engenharia de Software, como as de Pressman (2001), Sommerville (2003), Beck (1999) e dados empíricos como os relatórios do Standish Group (1995). O desenvolvimento do trabalho está estruturado em três capítulos principais, além desta introdução e da conclusão.

O Capítulo 1 apresenta os fundamentos da Engenharia de Software, explorando sua evolução histórica, definições e objetivos, com apoio teórico de autores como Sommerville (2003) e Paula Filho (2009). O Capítulo 2 trata das metodologias tradicionais de desenvolvimento, com ênfase no modelo

clássico ou sequencial, detalhando sua estrutura, aplicação e críticas, como as formuladas por Brooks (1987) e Gilb (1999). Já o Capítulo 3 aborda as metodologias ágeis, com destaque para o Manifesto Ágil, o Extreme Programming e suas práticas fundamentais, como planejamento, feedback, simplicidade, programação em pares e integração contínua.

Por fim, na conclusão, são retomadas as principais diferenças entre os dois modelos, discutindo-se as condições ideais para aplicação de cada um e destacando- se a importância de adaptação metodológica às características específicas de cada projeto. Com isso, espera-se que este estudo contribua para o entendimento crítico e atualizado sobre as metodologias de desenvolvimento de software, fortalecendo a capacidade de escolha consciente e estratégica por parte de profissionais e organizações da área.

2 DESENVOLVIMENTO

2.1 ENGENHARIA DE SOFTWARE

De acordo com Sommerville (2007, p. 5), o software compreende tanto os programas de computador quanto a documentação vinculada a eles. Esses produtos podem ser desenvolvidos sob encomenda para um cliente específico ou para atender a um público mais amplo. Pressman (1995, p. 12), por sua vez, conceitua software como um conjunto de instruções que, ao serem executadas, proporcionam determinada funcionalidade e desempenho. Com base nisso, pode-se compreender o software como um conjunto de algoritmos interpretados pela máquina com a finalidade de executar uma tarefa definida, acompanhado de sua devida documentação, podendo ou não ter uma aplicação comercial.

Compreender não apenas o que é software, mas também suas propriedades, é fundamental. As características do software são indispensáveis para se entender o processo de desenvolvimento e manutenção. Conforme destaca Pressman (1995), por ser um componente lógico e imaterial, o software não deve ser tratado como produtos manufaturados tangíveis, pois sua criação ocorre em ambiente virtual e não está sujeito a desgastes físicos. Entretanto, como qualquer produto, o software precisa de melhorias constantes e de manutenções regulares ao longo de seu ciclo de vida.

Mesmo não sofrendo deterioração ambiental, o software requer atualizações que podem introduzir falhas. Pressman (1995) ilustra essa dinâmica por meio de uma curva de falhas, na qual os picos representam as alterações feitas no software durante os processos de manutenção. Cada modificação pode gerar novos erros e, antes que a taxa de falhas retorne ao nível inicial, uma nova mudança pode ocorrer, provocando mais um aumento na curva. Ao longo do tempo, essa sequência contínua de modificações pode levar a uma elevação gradual do nível mínimo da curva, indicando um desgaste decorrente das manutenções (PRESSMAN, 1995).

Diante da constatação de que o software também se deteriora, ainda que por vias diferentes das de produtos físicos, surgiu a necessidade de uma área específica de estudo voltada à melhoria do processo de

criação e comercialização de softwares. Assim, torna-se essencial organizar o processo de desenvolvimento para garantir a eficácia e qualidade do produto final. Paula Filho (2009, p. 5) observa que a Engenharia de Software tem como foco principal o software enquanto produto. Estão fora de seu escopo programas criados apenas como passatempo pelos desenvolvedores ou pequenas soluções descartáveis, elaboradas para resolver problemas pontuais e pessoais. A Engenharia de Software, portanto, foca em aplicações que exigem maior complexidade e padrão de qualidade, voltadas à distribuição e uso comercial.

O conceito de Engenharia de Software foi introduzido inicialmente por Friedrich Ludwig Bauer durante uma conferência na década de 1960 que discutia a chamada “crise do software”. Segundo Pressman (1995), Bauer definiu essa engenharia como o uso de princípios sólidos de engenharia para possibilitar, de forma econômica, o desenvolvimento de softwares confiáveis e eficientes, capazes de operar em sistemas reais.

Diversas definições surgiram desde então, mas a proposta de Sommerville (2007, p. 5) se destaca pela clareza e abrangência, ao definir Engenharia de Software como uma disciplina voltada a todos os aspectos do processo de produção de software, desde sua concepção até as atividades de manutenção após a implantação. Para sistematizar esse processo de desenvolvimento, foram criados os chamados modelos de processo de software, que estruturam em etapas a criação e a manutenção dos programas (PRESSMAN, 1995). Esses modelos se dividem entre os modelos prescritivos (ou tradicionais) e as metodologias ágeis.

2.2 METODOLOGIAS TRADICIONAIS

As metodologias tradicionais de desenvolvimento de software, frequentemente denominadas metodologias "pesadas" ou "orientadas à documentação", surgiram em um cenário tecnológico bastante distinto do que se observa atualmente. Essas abordagens foram concebidas em uma época em que o processo de desenvolvimento era centrado em mainframes, com terminais considerados burros, ou seja, dispositivos sem capacidade de processamento próprio Royce (1970). Nesse contexto, o acesso a computadores era restrito, os recursos eram escassos e não existiam ferramentas modernas de apoio ao desenvolvimento, como os depuradores (debuggers), analisadores de código ou ambientes integrados de desenvolvimento (IDEs) que hoje são comuns.

Devido a essas limitações, os custos associados a modificações e correções em softwares eram extremamente elevados. Qualquer erro descoberto tardiamente no processo significava um grande retrabalho, o que motivava uma ênfase quase absoluta no planejamento detalhado e na documentação minuciosa antes mesmo que qualquer linha de código fosse escrita. O foco estava em prever, antecipadamente, todas as etapas do sistema, reduzindo ao máximo as incertezas e garantindo que os requisitos estivessem completamente definidos antes da fase de implementação. Royce (1970)

Nesse cenário, consolidou-se o chamado modelo clássico, também conhecido como modelo cascata

(waterfall), proposto inicialmente por Royce em 1970. Este modelo tornou-se a principal representação das metodologias tradicionais e, apesar das transformações pelas quais passou o desenvolvimento de software ao longo das décadas, ainda é utilizado em determinados tipos de projetos, especialmente naqueles que envolvem requisitos estáveis e bem compreendidos desde o início. A estrutura sequencial do modelo clássico – composta por fases rígidas e sucessivas como levantamento de requisitos, análise, projeto, implementação, testes e manutenção – reflete essa mentalidade de controle rigoroso e previsibilidade. Royce (1970)

Atualmente, embora os métodos ágeis tenham ganhado espaço significativo devido à sua flexibilidade e capacidade de adaptação às mudanças, as metodologias tradicionais continuam sendo relevantes em contextos que exigem alto grau de formalidade, documentação completa e conformidade com normas e regulamentações. Por isso, é importante compreendê-las historicamente e metodologicamente, uma vez que formam a base sobre a qual muitas das práticas modernas foram construídas.

2.3 MODELO CLÁSSICO

O modelo Clássico, também conhecido como modelo Sequencial ou em Cascata, foi o primeiro processo estruturado de desenvolvimento de software a ser formalmente divulgado, conforme Pressman (2001). Desde sua criação, esse modelo tem sido amplamente empregado, especialmente em contextos onde há uma clareza inicial sobre os requisitos do sistema. Ele é caracterizado por um fluxo linear de etapas, em que o término de uma fase condiciona o início da próxima. Cada fase deve ser finalizada com uma documentação específica que, após aprovada, autoriza a continuidade para o estágio seguinte.

De forma geral, o modelo Clássico compreende as seguintes fases: levantamento e definição de requisitos, elaboração do projeto de software, codificação e testes unitários, integração e testes de sistema, seguidos pelas etapas de operação e manutenção. A principal crítica a esse modelo reside em sua estrutura rígida, que dificulta a incorporação de mudanças ao longo do projeto — algo bastante comum no desenvolvimento de software. Por isso, recomenda-se sua utilização apenas em situações nas quais os requisitos estejam totalmente definidos e bem compreendidos desde o início do processo. Pressman (2001)

Durante muitos anos, o modelo Clássico dominou o cenário do desenvolvimento de software, especialmente até o início dos anos 1990. Contudo, diversos especialistas e pesquisadores começaram a apontar suas limitações. Fred Brooks, por exemplo, em seu renomado artigo *No Silver Bullet: Essence and Accidents of Software Engineering*, publicado em 1987, argumenta que é inviável especificar completamente um software antes do início da sua implementação Brooks (1987). Tom Gilb, outro nome de destaque na área, também critica a rigidez do modelo Clássico, recomendando o desenvolvimento incremental como uma abordagem mais segura e eficaz, especialmente para projetos de maior porte, pois oferece menor risco e maior probabilidade de êxito Gilb (1999).

Dados levantados pelo Standish Group em 1995, com base na análise de 8.380 projetos de software, evidenciam os problemas práticos relacionados à aplicação do modelo Clássico. Os números são alarmantes: apenas 16,2% dos projetos foram concluídos dentro do prazo, do orçamento previsto e com todas as funcionalidades inicialmente planejadas. Cerca de 31% foram cancelados antes de sua conclusão, e 52,7% foram finalizados, mas extrapolando prazos, custos ou com funcionalidades incompletas. Entre os projetos que não atenderam às expectativas iniciais, verificou- se uma média de atraso de 222% e um aumento médio de 189% no custo. Além disso, desses projetos entregues com atraso e sobrecusto, apenas 61% das funcionalidades planejadas foram efetivamente implementadas.

Mesmo os projetos que conseguiram cumprir prazos e orçamentos apresentaram problemas de qualidade, o que, segundo o estudo, pode estar ligado à pressão excessiva sobre os desenvolvedores — uma situação que, de acordo com a pesquisa, quadruplica a incidência de erros no software. Diante de tais resultados, ficou evidente que o modelo Clássico apresentava sérias deficiências em projetos mais complexos e dinâmicos. A recomendação final do estudo do Standish Group (1995) foi a adoção de modelos incrementais de desenvolvimento, por serem mais adaptáveis às mudanças e mais eficientes na redução das falhas observadas.

2.4 METODOLOGIAS ÁGEIS

O termo “Metodologias Ágeis” ganhou notoriedade a partir de 2001, quando dezessete especialistas em desenvolvimento de software, representando métodos como Scrum Schwaber e Beedle (2002), Extreme Programming – XP Beck (1999) e outros, reuniram-se para definir princípios comuns a essas abordagens. Esse encontro resultou na criação da Aliança Ágil e na formulação do chamado *Manifesto Ágil* Agile Manifesto (2004), um documento que passou a nortear práticas de desenvolvimento mais flexíveis e colaborativas.

O *Manifesto Ágil* propõe uma mudança de foco em relação aos modelos tradicionais, ao valorizar mais as pessoas e suas interações do que os processos e ferramentas; a entrega de software funcional em vez de documentação excessiva; a colaboração com o cliente em vez da rigidez contratual; e a capacidade de responder rapidamente a mudanças em lugar do apego a planos fixos.

Vale destacar que o manifesto não ignora a importância dos elementos relegados ao segundo plano, como processos, ferramentas, documentação e contratos. Ele apenas ressalta que, no contexto ágil, esses elementos não devem se sobrepor à flexibilidade, à comunicação eficiente e à entrega contínua de valor.

Essa filosofia se adapta melhor à realidade de pequenas e médias empresas, que muitas vezes lidam com mudanças frequentes e recursos limitados. Entre as metodologias ágeis mais conhecidas destaca-se a *Extreme Programming* (XP), proposta por Beck (1999), voltada especialmente para equipes pequenas e médias que trabalham com requisitos instáveis e em constante evolução. A XP se diferencia por promover

feedback contínuo, ciclos incrementais e forte incentivo à comunicação entre os membros da equipe e com o cliente.

O projeto C3 da Chrysler foi um marco na história da XP. Após sucessivos fracassos utilizando metodologias tradicionais, a aplicação da XP resultou na entrega do sistema em pouco mais de um ano [Highsmith et al. (2000)]. A XP causou impacto no mundo do desenvolvimento por propor práticas que, isoladamente, podem parecer controversas, mas que, quando aplicadas de forma conjunta e sinérgica, se mostram eficazes e inovadoras. O objetivo central da XP é garantir entregas rápidas, satisfação do cliente e maior assertividade nas estimativas, além de proporcionar um ambiente de trabalho colaborativo e produtivo.

Beck (1999) define quatro valores fundamentais para a XP: comunicação, simplicidade, feedback e coragem. A comunicação visa fortalecer o relacionamento entre desenvolvedores e clientes, privilegiando conversas diretas e evitando e-mails ou telefonemas sempre que possível. A simplicidade, por sua vez, orienta o desenvolvimento de códigos objetivos, sem funcionalidades desnecessárias ou voltadas a requisitos futuros incertos. Implementar apenas o que é necessário no presente, com possibilidade de ajustar futuramente, é mais eficaz do que tentar prever tudo de antemão.

O feedback constante é outro pilar da XP. Ele se manifesta tanto nos testes automatizados que avaliam continuamente o código, quanto nas entregas frequentes de versões funcionais do software para que o cliente possa acompanhar e sugerir melhorias. Com isso, eventuais falhas ou equívocos são detectados precocemente e corrigidos com agilidade, aumentando as chances de o produto final estar de acordo com as expectativas do cliente.

Já a coragem é necessária para sustentar os outros três valores. Nem todos os profissionais estão habituados à comunicação direta ou à exposição constante a críticas e mudanças. Ter coragem, nesse contexto, significa estar aberto à melhoria contínua, aceitar alterações de requisitos e não temer revisões frequentes de código. Além dos valores, a XP se apoia em doze práticas essenciais [Beck (1999)].

Entre elas, está o planejamento, que define as prioridades do projeto com base em requisitos atuais, e não em previsões futuras. O cliente participa ativamente das decisões sobre escopo, entregas e versões, enquanto os desenvolvedores definem prazos e cronogramas. As entregas frequentes fazem parte dessa lógica, permitindo que versões reduzidas, porém funcionais, sejam disponibilizadas mensalmente, favorecendo o feedback contínuo.

A prática da metáfora permite descrever o funcionamento do software em termos simples, sem jargões técnicos, facilitando o entendimento por todos os envolvidos. O projeto simples é uma diretriz para evitar códigos complexos ou recursos prematuros. Os testes são desenvolvidos antes da codificação, garantindo que o sistema funcione como esperado desde os estágios iniciais.

A programação em pares, outra característica marcante da XP, consiste em dois desenvolvedores

trabalhando juntos no mesmo computador: um escreve o código enquanto o outro revisa em tempo real, sugerindo melhorias e identificando falhas. Essa interação constante favorece o aprendizado mútuo e a qualidade do código. Já a refatoração visa aprimorar o design do software, tornando-o mais limpo e eficiente sem comprometer suas funcionalidades.

Na propriedade coletiva, o código pertence a toda a equipe. Qualquer membro pode modificá-lo, desde que realize os testes necessários, promovendo responsabilidade compartilhada e facilitando a continuidade do projeto, mesmo diante de eventuais saídas de integrantes. A integração contínua garante que o sistema seja atualizado e testado diversas vezes ao dia, minimizando conflitos de código e falhas. A XP também preza pela sustentabilidade do trabalho, propondo jornadas regulares de até 40 horas semanais. O excesso de horas é visto como sinal de falha no planejamento, e não como solução para atrasos. O envolvimento direto do cliente presente durante todo o processo assegura esclarecimento rápido de dúvidas e alinhamento constante com as expectativas. Por fim, o uso de código padrão facilita o entendimento coletivo da base de código, promovendo coesão e qualidade técnica.

Em síntese, a XP propõe uma forma revolucionária de desenvolver software, baseada em colaboração, simplicidade e adaptação contínua, oferecendo uma alternativa eficaz frente às limitações das abordagens tradicionais.

3 CONCLUSÃO

A conclusão deste trabalho ressalta a importância da comparação entre metodologias ágeis e tradicionais no ciclo de vida do software, evidenciando que não existe uma abordagem única que se destaque como a melhor para todos os cenários. As metodologias tradicionais, como o modelo em cascata, ainda têm sua relevância em contextos onde os requisitos são bem definidos e a documentação rigorosa é necessária. Elas proporcionam uma estrutura clara e sequencial que pode ser vantajosa em projetos de menor complexidade ou em ambientes regulados.

Por outro lado, as metodologias ágeis, representadas por abordagens como Extreme Programming e Scrum, se mostram mais adequadas para ambientes dinâmicos e colaborativos, onde a flexibilidade e a capacidade de adaptação às mudanças são cruciais. O Manifesto Ágil, com seus valores voltados para a comunicação, entrega contínua e colaboração com o cliente, reflete uma resposta às exigências contemporâneas do mercado, onde a inovação e a rapidez na entrega de valor são fundamentais.

A análise realizada neste trabalho confirma que a escolha da metodologia deve ser estratégica, levando em consideração fatores como o tipo de projeto, a estabilidade dos requisitos e o contexto organizacional. As metodologias ágeis, com sua ênfase na interação constante entre equipes e clientes, permitem uma maior agilidade e redução de riscos, promovendo um ambiente de trabalho mais produtivo e satisfatório.

Além disso, este estudo destaca a necessidade de um entendimento crítico sobre as metodologias de desenvolvimento, permitindo que profissionais e organizações façam escolhas mais informadas e alinhadas aos seus objetivos. A capacidade de adaptação às características específicas de cada projeto é essencial para maximizar a eficiência e a qualidade do produto final, contribuindo assim para a satisfação do cliente e o sucesso do empreendimento.

Em suma, enquanto as metodologias tradicionais ainda têm seu espaço, as abordagens ágeis têm se destacado como uma resposta eficaz às demandas atuais do desenvolvimento de software. A evolução contínua das práticas de engenharia de software exige que os profissionais estejam sempre atualizados e prontos para adotar a metodologia que melhor se encaixe nas necessidades do projeto em questão, garantindo assim a entrega de soluções de qualidade e que realmente atendam às expectativas do mercado.

REFERÊNCIAS

AGILE MANIFESTO. Disponível em: http://agilemanifesto.org/. Acesso em: 30 Jun. 2025.

BECK, K. Programação extrema explicada. Bookman, 1999.

BROOKS, F. No silver bullet: essence and accidents of software engineering. In: Proc. IFIP. IEEE CS Press, 1987. p. 1069-1076. Reimpresso em IEEE Computer, abr. 1987. p. 10-19.

CHARETTE, R. Fair fight? Agile versus heavy methodologies. Cutter Consortium E- project Management Advisory Service, v. 2, n. 13, 2001.

COCKBURN, A.; HIGHSMITH, J. Agile software development: the business of innovation. IEEE Computer, set. 2001. p. 120-122.

GILB, T. Principles of software engineering management. Addison-Wesley, 1988.

HIGHSMITH, J.; ORR, K.; COCKBURN, A. Extreme programming. E-Business Application Delivery, fev. 2000. p. 4-17.

PRESSMAN, R. Engenharia de software. McGraw-Hill, 2001.

ROYCE, W. W. Managing the development of large software systems: concepts and techniques. In: Proc. IEEE Westcon. Los Angeles, CA, 1970.

SCHWABER, K.; BEEDLE, M. Agile software development with Scrum. New Jersey: Prentice-Hall, 2002.

SOMMERVILLE, I. Engenharia de software. São Paulo: Addison-Wesley, 2003.

STANDISH GROUP. CHAOS report. 586 Olde Kings Highway. Dennis, MA 02638, USA, 1995.

XPLANNER. Disponível em: http://www.xplanner.org/. Acesso em: 30 Jun. 2025.

KOSCIANSKI, André; SOARES, Michel dos Santos. Qualidae de Software: Aprenda as metodologias e técnicas mais modernas para o desenvolvimento de software.
2. ed. São Paulo: Novatec Editora, 2007.

PAULA FILHO, Wilson de Pádua. Engenharia de software: fundamentos, métodos e padrões. 2. ed. Rio de Janeiro: LTC, 2003.

PAULA FILHO, Wilson de Pádua. Engenharia de software: fundamentos, métodos e padrões. 3. ed. Rio de Janeiro: LTC, 2009.

PRESSMAN, Roger S. Engenharia de software. Trad. José Carlos Barbosa dos Santos. 3. ed. São Paulo: Makron Books, 1995.

PRESSMAN, Roger S. Engenharia de software. Trad. Rosângela Delloso Penteado.
6. ed. São Paulo: McGraw-Hill, 2006.

PRESSMAN, Roger S. Engenharia de software: uma abordagem profissional. Trad. Ariovaldo Griesi. 7. ed. São Paulo: Bookman, 2011.

PRIKLADNICKI, Rafael; WILLI, Renato; MILANI, Fabiano. Métodos Ágeis Para Desenvolvimento De Software. Porto Alegre: Bookman, 2014.

SOARES, Michel dos Santos. Comparação entre metodologias ágeis e tradicionais para odesenvolvimento de Software. 2004. Disponível em: <http://www.dcc.ufla.br/infocomp/index.php/INFOCOMP/article/view/68/53>. Acesso em: 13 nov. 2015.

SOMMERVILLE, Ian. Engenharia de software. Trad. Selma Shin Shimizu Melnikoff et al. 8. ed. São Paulo: Pearson Addison-Wesley, 2007.

SOMMERVILLE, Ian. Engenharia de software. Trad. Ivan Bosnic e Kalinka G. 9. ed. São Paulo: Pearson Prentice Hall, 2011.