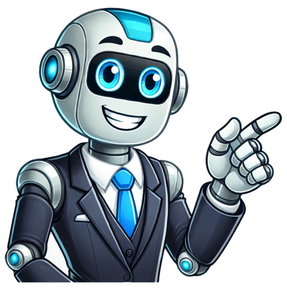
Continue



Livro de engenharia de software

Resultado da busca Categoria: Engenharia de Software Foram encontradas 40 publicações Mostrando de 1 a 10 Engenharia de Software para Cientistas de Dados Catherine Nelson ISBN impresso: 978-85-7522-917-0 ISBN ebook: 978-85-7522-918-7 Ano: 2024 Páginas: 288 Preço: R$ 93,00 Disponível também em formato ebook.Leia mais > Tidy First? Kent Beck ISBN impresso: 978-85-7522-899-9 ISBN ebook: 978-85-7522-900-2 Ano: 2024 Páginas: 112 Preço: R$ 62,00 Disponível também em formato ebook.Leia mais > Lean UX – 3ª Edição Jeff Gothelf / Josh Seiden ISBN impresso: 978-85-7522-833-3 ISBN ebook: 978-85-7522-834-0 Ano: 2022 Páginas: 288 Preço: R$ 97,00 Disponível também em formato ebook.Leia mais > Criando Microsserviços - 2ª Edição Sam Newman ISBN impresso: 978-65-86057-88-1 ISBN ebook: 978-65-86057-89-8 Ano: 2022 Páginas: 688 Preço: R$ 159,00 Disponível também em formato ebook.Leia mais > Construindo uma Carreira em Software Daniel Heller ISBN impresso: 978-65-86057-69-0 ISBN ebook: 978-65-86057-70-6 Ano: 2021 Páginas: 304 Preço: R$ 89,00 Disponível também em formato ebook.Leia mais > Linguagens Formais Exercícios e Soluções Marcus Vinicius Midena Ramos ISBN impresso: 978-65-86057-59-1 Ano: 2021 Páginas: 336 Preço: R$ 85,00 Leia mais > SQL em 10 Minutos por Dia Ben Forta ISBN impresso: 978-65-86057-44-7 ISBN ebook: 978-65-86057-45-4 Ano: 2021 Páginas: 272 Preço: R$ 79,00 Disponível também em formato ebook.Leia mais > Refatoração – 2ª Edição Martin Fowler ISBN impresso: 978-85-7522-724-4 ISBN ebook: 978-85-7522-725-1 Ano: 2020 Páginas: 456 Preço: R$ 129,00 Disponível também em formato ebook.Leia mais > Design de Microsserviços com Django Akos Hochrein ISBN impresso: 978-65-86057-06-5 ISBN ebook: 978-65-8605-707-2 Ano: 2020 Páginas: 144 Preço: R$ 62,00 Disponível também em formato ebook.Leia mais > Problemas Clássicos de Ciência da Computação com Python David Kopec ISBN impresso: 978-85-7522-805-0 ISBN ebook: 978-85-7522-806-7 Ano: 2019 Páginas: 272 Preço: R$ 79,00 Disponível também em formato ebook.Leia mais >1 2 3 4 Próxima >> Share — copy and redistribute the material in any medium or format for any purpose, even commercially. Adapt — remix, transform, and build upon the material for any purpose, even commercially. The licensor cannot revoke these freedoms as long as you follow the license terms. Attribution — You must give appropriate credit , provide a link to the license, and indicate if changes were made . You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use. ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits. You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation . No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material. Página principal do livro Compre na Amazon ou UmLivro Veja também nosso curso de extensão a distância, com certificados emitidos pelo DCC/UFMG. In software development, perfect is a verb, not an adjective. There is no perfect process. There is no perfect design. There are no perfect stories. You can, however, perfect your process, your design, and your stories. – Kent Beck Este capítulo inicia com uma apresentação sobre a importância de processos de software (Seção 2.1). Em seguida, discutimos questões gerais e preliminares sobre processos ágeis de desenvolvimento de software (Seção 2.2), incluindo uma apresentação sobre o contexto histórico que motivou o surgimento desse tipo de processo. As próximas seções tratam de três métodos ágeis: Extreme Programming (Seção 2.3), Scrum (Seção 2.4) e Kanban (Seção 2.5). Depois, temos uma seção dedicada a discutir quando métodos ágeis não são recomendados (Seção 2.6). Por fim, na Seção 2.7, discutimos alguns processos tradicionais, principalmente o Processo Unificado. Importância de Processos A produção de um carro em uma fábrica de automóveis segue um processo bem definido. Sem estender a explicação, primeiro, as chapas de aço são cortadas e prensadas, para ganhar a forma de portas, tetos e capôs. Depois, o carro é pintado e instalam-se painel, bancos, cintos de segurança e a fiação. Por fim, instala-se a parte mecânica, incluindo motor, suspensão e freios. Assim como carros, software também é produzido de acordo com um processo, embora certamente menos mecânico e mais dependente de esforço intelectual. Um processo de desenvolvimento de software define um conjunto de passos, tarefas, eventos e práticas que devem ser seguidos por desenvolvedores de software, na produção de um sistema. Alguns críticos de processos de software costumam fazer a seguinte pergunta: por que eu preciso seguir um processo? E complementam, perguntando também o seguinte: Qual processo Linus Torvalds usou na implementação do sistema operacional Linux? Ou que Donald Knuth usou na implementação do formatador de textos TeX? Na verdade, a segunda parte da pergunta não faz muito sentido, pois tanto o Linux (no seu início) e o TeX são projetos individuais, liderados por um único desenvolvedor. Nesses casos, a adoção de um processo é menos importante. Ou, dizendo de outra forma, o processo em tais projetos é pessoal, composto pelos princípios, práticas e decisões tomadas pelo seu único desenvolvedor; e que terão impacto apenas sobre ele mesmo. Porém, os sistemas de software atuais são por demais complexos para serem desenvolvidos por uma única pessoa. Por isso, casos de sistemas desenvolvidos por heróis serão cada vez mais raros. Na prática, os sistemas modernos — que nos interessam neste livro — são desenvolvidos em equipes. E essas equipes, para produzir software com qualidade e produtividade, precisam de um ordenamento, mesmo que mínimo. Por isso, empresas dão tanto valor a processos de software. Eles são o instrumento de que as empresas dispõem para coordenar, motivar, organizar e avaliar o trabalho de seus desenvolvedores, de forma a garantir que eles trabalhem com produtividade e produzam sistemas alinhados com os objetivos da organização. Sem um processo — mesmo que simplificado e leve, como os processos ágeis que estudaremos neste capítulo — existe o risco de que os times de desenvolvimento passem a trabalhar de forma descoordenada, gerando produtos sem valor para o negócio da empresa. Por fim, processos são importantes não apenas para a empresa, mas também para os desenvolvedores, pois permitem que eles tomem consciência das tarefas e resultados que se esperam deles. Sem um processo, os desenvolvedores podem se sentir perdidos, trabalhando de forma errática e sem alinhamento com os demais membros do time. Neste capítulo, vamos estudar alguns processos de software. Na verdade, no Capítulo 1 já comentamos sobre Processos Waterfall e Ágeis. Na próxima seção, vamos retomar essa discussão e, em seguida, descrever alguns métodos de desenvolvimento de software. Manifesto Ágil Os primeiros processos de desenvolvimento de software — do tipo Waterfall, propostos ainda na década de 70 — eram estritamente sequenciais, começando com uma fase de especificação de requisitos até chegar às fases finais de implementação, testes e manutenção do sistema. Se considerarmos o contexto histórico, essa primeira visão de processo era natural, visto que projetos de Engenharia tradicional também são sequenciais e precedidos de um planejamento detalhado. Todas as fases também geram documentações detalhadas do produto que está sendo desenvolvido. Por isso, nada mais natural que a nascente Engenharia de Software se espelhasse nos processos de áreas mais tradicionais, como Engenharia Eletrônica, Civil, Mecânica, Aeronáutica, etc. No entanto, após cerca de uma década, começou-se a perceber que software é diferente de outros produtos de Engenharia. Essa percepção foi ficando clara devido aos problemas frequentes enfrentados por projetos de software nas décadas de 70 a 90. Por exemplo, os cronogramas e orçamentos desses projetos não eram obedecidos. Não raro, projetos inteiros eram cancelados, após anos de trabalho, sem entregar um sistema funcional para os clientes. Em 1994, um relatório produzido pela empresa de consultoria Standish Group revelou informações mais detalhadas sobre os projetos de software da época. Por exemplo, o relatório, que ficou conhecido pelo sugestivo nome de CHAOS Report (link), mostrou que mais de 55% dos projetos estourava os prazos planejados entre 51% e 200%; pelo menos 12% estouravam os prazos acima de 200%, conforme mostra o próximo gráfico. CHAOS Report (1994): percentual de projetos que estourava seus prazos (para cada faixa de estouro). Os resultados em termos de custos não eram mais animadores: quase 40% dos projetos ultrapassava o orçamento entre 51% e 200%, como mostra o seguinte gráfico: CHAOS Report (1994): percentual de projetos que estourava seus orçamentos (para cada faixa de estouro). Em 2001, um grupo de profissionais da indústria se reuniu na cidade de Snowbird, no estado norte-americano de Utah, para discutir e propor uma alternativa aos processos do tipo Waterfall que então predominavam. Essencialmente, eles passaram a defender que software é diferente de produtos tradicionais de Engenharia. Por isso, software também demanda um processo de desenvolvimento diferente. Por exemplo, os requisitos de um software mudam com frequência, mais do que os requisitos de um computador, de um avião ou de uma ponte. Além disso, os clientes frequentemente não têm uma ideia precisa do que querem. Ou seja, corre-se o risco de projetar por anos um produto que depois de pronto não será mais necessário, ou porque o mundo mudou ou porque os planos e as necessidades dos clientes mudaram. Eles diagnosticaram ainda problemas nos documentos prescritos por processos Waterfall, incluindo documentos de requisitos, fluxogramas, diagramas, etc. Esses documentos eram detalhados, pesados e extensos. Assim, rapidamente se tornavam obsoletos, pois quando os requisitos mudavam os desenvolvedores não propagavam as alterações para a documentação, mas apenas para o código. Então eles decidiram lançar as bases para um novo conceito de processo de software, as quais foram registradas em um documento que chamaram de Manifesto Ágil. Por ser curto, iremos reproduzir o texto do manifesto: Por meio deste trabalho, passamos a valorizar: Indivíduos e interações, mais do que processos e ferramentas Software em funcionamento, mais do que documentação abrangente Colaboração com o cliente, mais do que negociação de contratos Resposta a mudanças, mais do que seguir um plano. A característica principal de processos ágeis é a adoção de ciclos curtos e iterativos de desenvolvimento, por meio dos quais um sistema é implementado de forma gradativa; começando por aquilo que é mais urgente para o cliente. De início, implementa-se uma primeira versão do sistema, com as funcionalidades que segundo o cliente são para ontem, isto é, possuem prioridade máxima. Em seguida, essa versão é validada pelo cliente. Se ela for aprovada, um novo ciclo — ou iteração — inicia-se, com mais algumas funcionalidades, também priorizadas pelos clientes. Normalmente, esses ciclos são curtos, com duração de um mês, talvez até um pouco menos. Assim, o sistema vai sendo construído de forma incremental, sendo cada incremento devidamente aprovado pelos clientes. O desenvolvimento termina quando o cliente decide que todos os requisitos estão implementados. As próximas figuras comparam desenvolvimento em Waterfall e Ágil. Desenvolvimento usando um Processo Waterfall. O sistema fica pronto apenas no final. Desenvolvimento usando um Processo Ágil. A cada iteração (representada pelos retângulos) gera-se um incremento no sistema (S++), que já pode ser validado e testado pelos usuários finais. No entanto, a figura anterior pode sugerir que, em desenvolvimento ágil, cada iteração é um mini-waterfall, incluindo todas as fases de um processo Waterfall. Isso não é verdadeiro; em geral, as iterações em métodos ágeis não são um pipeline de tarefas, como em Waterfall (mais sobre isso nas próximas seções). A figura também pode sugerir que ao final de cada iteração tem-se que colocar um sistema em produção, para uso pelos usuários finais. Isso também não é verdade. De fato, o objetivo é entregar um sistema funcional, isto é, que realize tarefas úteis. Porém, a decisão de colocá-lo em produção envolve outras variáveis, como riscos para o negócio da empresa, disponibilidade de servidores, campanhas de marketing, elaboração de manuais, treinamento de usuários, etc. Outras características de processos ágeis incluem: Menor ênfase em documentação, ou seja, apenas o essencial deve ser documentado. Menor ênfase em planos detalhados, pois muitas vezes nem o cliente, nem os Engenheiros de Software têm, no início de um projeto, uma ideia clara dos requisitos que devem ser implementados. Esse entendimento vai surgir ao longo do caminho, à medida que incrementos de produto sejam produzidos e validados. Em outras palavras, o importante em desenvolvimento ágil é conseguir avançar, mesmo em ambientes com informações imperfeitas, parciais e sujeitas a mudanças. Inexistência de uma fase dedicada a design (big design up front). Em vez disso, o design também é incremental. Ele evolui à medida que o sistema vai nascendo, ao final de cada iteração. Desenvolvimento em times pequenos, com cerca de uma dezena de desenvolvedores. Ou, em outras palavras, times que possam ser alimentados com duas pizzas, conforme popularizado pelo CEO da Amazon, Jeff Bezos. Ênfase em novas práticas de desenvolvimento (pelo menos, para o início dos anos 2000), como programação em pares, testes automatizados e integração contínua. Devido a essas características, processos ágeis são considerados processos leves, com poucas prescrições e documentos. No entanto, as características acima são genéricas e abrangentes; por isso alguns métodos foram propostos para ajudar desenvolvedores a adotar os princípios ágeis, de forma mais concreta. O interessante é que todos eles foram propostos, pelo menos na primeira versão, antes do encontro de Utah, em 2001, que lançou o Manifesto Ágil. Neste capítulo, vamos estudar três métodos ágeis: Extreme Programming (XP), proposto por Kent Beck, em um livro lançado em 1999 (link). Uma segunda edição do livro, incluindo uma grande revisão, foi lançada em 2004. Neste capítulo, vamos nos basear nessa edição mais recente. Scrum, proposto por Jeffrey Sutherland e Ken Schwaber, em um artigo publicado em 1995 (link). Kanban, cujas origens remontam a um sistema de controle de produção que começou a ser usado nas fábricas da Toyota, ainda na década de 50 (link). Nos últimos 10 anos, Kanban tem sido gradativamente adaptado para uso no desenvolvimento de software. Aprofundamento: Neste livro, usamos os termos processos e métodos. Processo é o conjunto de passos, etapas e tarefas que se usa para construir um software. Toda organização usa um processo para desenvolver seus sistemas, o qual pode ser ágil ou waterfall, por exemplo. Ou, talvez, esse processo pode ser caótico. Porém, o ponto que queremos reforçar é que sempre existe um processo. Já método, no nosso contexto, define e especifica um determinado processo de desenvolvimento (a palavra método tem sua origem no grego, onde significa caminho para se chegar a um objetivo). Assim, XP, Scrum e Kanban são métodos ágeis ou, de modo mais extenso, são métodos que definem práticas, atividades, eventos e técnicas compatíveis com princípios ágeis de desenvolvimento de software. Aproveitando que estamos tratando de definições, frequentemente usa-se também o termo metodologia quando se fala de processos de software. Por exemplo, é comum ver referências a metodologias para desenvolvimento de software, metodologias ágeis, metodologia orientada a objetos, etc. A palavra metodologia, no sentido estrito, denota o ramo da lógica que se ocupa dos métodos das diferentes ciências, segundo o Dicionário Houaiss. No entanto, a palavra também pode ser usada como sinônimo de método, segundo o mesmo dicionário. Apesar disso, neste livro evitamos usar o termo metodologia e tentamos empregar sempre o termo método. Aviso: Todo método de desenvolvimento deve ser entendido como um conjunto de recomendações; cabe a uma organização analisar cada uma e decidir se ela faz sentido no seu contexto. Como resultado, a organização pode ainda decidir por adaptar essas recomendações para atender às suas necessidades. Logo, provavelmente, não existem duas organizações que seguem exatamente o mesmo processo de desenvolvimento, mesmo que elas digam que estão desenvolvendo usando Scrum, por exemplo. Mundo Real: O sucesso e impacto de processos ágeis foi impressionante. Hoje, a grande maioria das empresas que desenvolvem software, independente de seu tamanho ou do foco de seu negócio, usam princípios ágeis, em maior ou menor escala. Para citar alguns dados, em 2018, o Stack Overflow survey incluiu uma pergunta sobre o método de desenvolvimento mais usado pelos respondentes (link). Essa pergunta recebeu 57 mil respostas de desenvolvedores profissionais e a grande maioria mencionou métodos ou práticas ágeis, incluindo aquelas que vamos estudar neste capítulo, como Scrum (63% das respostas), Kanban (36%) e Extreme Programming (16%). Apenas 15% dos participantes marcaram Waterfall como resposta. Extreme Programming Segundo seu autor, XP é um método leve recomendado para desenvolver software com requisitos vagos ou sujeitos a mudanças; isto é, basicamente sistemas comerciais, na classificação que adotamos no Capítulo 1. Sendo um método ágil, XP possui todas as características que mencionamos na seção anterior, isto é: adota ciclos curtos e iterativos de desenvolvimento, concede menos ênfase para documentação e para planos detalhados, propõe que o design de um sistema também seja definido de forma incremental e sugere que as equipes de desenvolvimento sejam pequenas. Porém, XP não é um método prescritivo, que define um passo a passo detalhado para construção de software. Em vez disso, XP é definido por meio de um conjunto de valores, princípios e práticas de desenvolvimento. Ou seja, XP é inicialmente definido de forma abstrata, usando-se de valores e princípios que devem fazer parte da cultura e dos hábitos de times de desenvolvimento de software. Depois, esses valores e princípios são concretizados em uma lista de práticas de desenvolvimento. Frequentemente, quando decidem adotar XP, desenvolvedores e organizações concentram-se nas práticas. Porém, os valores e princípios são componentes chaves do método, pois são eles que dão sentido às práticas propostas em XP. Sendo mais claro, se uma organização não está preparada para trabalhar no modelo mental de XP — representado pelos seus valores e princípios — recomenda-se também não adotar suas práticas. Neste capítulo, vamos primeiro apresentar os valores e princípios de XP. Veja uma lista deles a seguir: Valores: comunicação, simplicidade, feedback, coragem, respeito e qualidade de vida. Princípios: humanidade, economicidade, benefícios mútuos, melhorias contínuas, falhas acontecem, baby steps e responsabilidade pessoal. Em seguida, vamos descrever as práticas. Para facilitar a explicação delas, resolvemos organizá-las em três grupos: práticas sobre o processo de desenvolvimento, práticas de programação e práticas de gerenciamento de projetos. Veja a seguir uma lista das práticas em cada grupo: Práticas sobre o Processo de Desenvolvimento: representante dos clientes, histórias dos usuários, iterações, releases, planejamento de releases, planejamento de iterações, planning poker, slack. Práticas de Programação: design incremental, programação pareada, desenvolvimento dirigido por testes (TDD), build automatizado, integração contínua. Práticas de Gerenciamento de Projetos: métricas, ambiente de trabalho, contratos com escopo aberto. Valores XP defende que o desenvolvimento de projetos de software seja norteado por três valores principais: comunicação, simplicidade e feedback. Na verdade, argumenta-se que esses valores são universais, para convívio humano. Ou seja, eles não servem apenas para guiar projetos de desenvolvimento, mas a própria vida em sociedade. Uma boa comunicação é importante em qualquer projeto, não apenas para evitar, mas também para aprender com erros. O segundo valor de XP é simplicidade, pois em todo sistema complexo e desafiador existem sistemas ou subsistemas mais simples, que às vezes não são considerados. Por último, existem riscos em todos os projetos de software: os requisitos mudam, a tecnologia muda, a equipe de desenvolvimento muda, o mundo muda, etc. Um valor que ajuda a controlar tais riscos é estar aberto ao feedback dos stakeholders, a fim de que correções de rota sejam implementadas o quanto antes. Em outras palavras, é difícil desenvolver o sistema de software certo em uma primeira e única tentativa. Frederick Brooks tem uma frase conhecida sobre esse fenômeno: Planeje-se para jogar fora partes de seu sistema, pois você fará isso. Por isso, feedback é um valor essencial para garantir que as partes ou versões que serão descartadas sejam identificadas o quanto antes, de forma a diminuir prejuízos e retrabalho. Além dos três valores mencionados, XP também defende outros valores, como coragem, respeito e qualidade de vida. Princípios Os valores que mencionamos são abstratos e universais. Por outro lado, as práticas que vamos mencionar mais adiante são procedimentos concretos e pragmáticos. Assim, para unir esses dois extremos, XP defende que projetos de software devem seguir um conjunto de princípios. A imagem que se apresenta é de um rio: de um lado estão os valores e de outro as práticas. Os princípios — que descreveremos agora — fazem o papel de uma ponte ligando esses dois lados. Alguns dos principais princípios de XP são os seguintes: Humanidade (humanity, em inglês). Software é uma atividade intensiva no uso de capital humano. O principal recurso de uma empresa de software não são seus bens físicos — computadores, prédios, móveis ou conexões de Internet, por exemplo — mas sim seus colaboradores. Um termo que reflete bem esse princípio é peopleware, que foi cunhado por Tom DeMarco, em um livro com o mesmo título (link). A ideia é que a gestão de pessoas — incluindo fatores como expectativas, crescimento, motivação, transparência e responsabilidade — é fundamental para o sucesso de projetos de software. Economicidade (economics, em inglês). Se por um lado, peopleware é fundamental, por outro lado software é uma atividade cara, que demanda a alocação de recursos financeiros consideráveis. Logo, tem-se que ter consciência de que o outro lado, isto é, quem está pagando as contas do projeto, espera resultados econômicos e financeiros. Por isso, na grande maioria dos casos, software não pode ser desenvolvido apenas para satisfazer a vaidade intelectual de seus desenvolvedores. Software não é uma obra de arte, mas algo que tem que gerar resultados econômicos, como defendido por esse princípio de XP. Benefícios Mútuos. XP defende que as decisões tomadas em um projeto de software têm que beneficiar múltiplos stakeholders. Por exemplo, o contratante do software deve garantir um bom ambiente de trabalho (peopleware); em contrapartida, a equipe deve entregar um sistema que agregue valor ao seu negócio (economicidade). Mais um exemplo: ao escrever testes um desenvolvedor se beneficia, pois eles ajudam a detectar bugs no seu código; mas testes também ajudam outros desenvolvedores, que futuramente terão mais segurança de que o seu código não vai introduzir regressões — isto é, bugs — em código que está funcionando. Um terceiro e último exemplo: refactoring é uma atividade que torna o código mais limpo e fácil de entender, tanto para quem o escreveu, como para quem futuramente terá que mantê-lo. A frase todo negócio tem que ser bom para os dois lados resume bem esse terceiro princípio de XP. Melhorias Contínuas (no livro de XP, o nome original é improvements): Como expressa a frase de Kent Beck que abre este capítulo, nenhum processo de desenvolvimento de software é perfeito. Por isso, é mais seguro trabalhar com um sistema que vai sendo continuamente aprimorado, a cada iteração, com o feedback dos clientes e de todos os membros do time. Pelo mesmo motivo, XP não recomenda investir um grande montante de tempo em um design inicial e completo. Em vez disso, o design do sistema também é incremental, melhorando a cada iteração. Por fim, as próprias práticas de desenvolvimento podem ser aprimoradas; para isso, o time deve reservar tempo para refletir sobre elas. Falhas Acontecem. Desenvolvimento de software não é uma atividade livre de riscos. Como discutido no Capítulo 1, software é uma das mais complexas construções humanas. Logo, falhas são esperadas em projetos de desenvolvimento de software. No contexto desse princípio, falhas incluem bugs, funcionalidades que não se mostraram interessantes para os usuários finais e requisitos não-funcionais que não estão sendo plenamente atendidos, como desempenho, usabilidade, privacidade, disponibilidade, etc. Evidentemente, XP não advoga que essas falhas devem ser acobertadas. Porém, elas não devem ser usadas para punir membros de um time. Pelo contrário, falhas fazem parte do jogo, se um time pretende entregar software com rapidez. Baby Steps. É melhor um progresso seguro, testado e validado, mesmo que pequeno, do que grandes implementações com riscos de serem descartadas pelos usuários. O mesmo vale para testes (que são úteis mesmo quando as unidades testadas são de menor granularidade), integração de código (é melhor integrar diariamente, do que passar pelo stress de fazer uma grande integração após semanas de trabalho) e refatorações (que devem ocorrer em pequenos passos, quando é mais fácil verificar que o comportamento do sistema está sendo preservado). Em resumo, o importante é garantir melhorias contínuas, não importando que sejam pequenas, desde que na direção correta. Essas pequenas melhorias são melhores do que grandes revoluções, as quais costumam não apresentar resultados positivos, pelo menos quando se trata de desenvolvimento de software. Responsabilidade Pessoal (que usamos como tradução para accepted responsibility). De acordo com esse princípio, desenvolvedores devem ter uma ideia clara de seu papel e responsabilidade na equipe. O motivo é que responsabilidade não pode ser transferida, sem que a outra parte a aceite. Por isso, XP defende que o engenheiro de software que implementa uma história — termo que o método usa para requisitos — deve ser também aquele que vai testá-la e mantê-la. Mundo Real: Um dos primeiros sistemas a adotar XP foi um sistema de folha de pagamentos da fabricante de automóveis Chrysler, chamado Chrysler Comprehensive Compensation (C3) (link). O projeto desse sistema começou no início de 1995 e, como não apresentou resultados concretos, ele foi reiniciado no ano seguinte, sob a liderança de Kent Beck. Outro membro conhecido da comunidade ágil, Martin Fowler, participou do projeto, como consultor. No desenvolvimento do sistema C3, foram usadas e testadas diversas ideias do método que poucos anos depois receberia o nome de XP. Práticas sobre o Processo de Desenvolvimento XP — como outros métodos ágeis — recomenda o envolvimento dos clientes com o projeto. Ou seja, além de desenvolvedores, os times incluem pelo menos um representante dos clientes, que deve entender do domínio do sistema que será construído. Uma das funções desse representante é escrever as histórias de usuário (user stories), que é o nome que XP dá para os documentos que descrevem os requisitos do sistema a ser implementado. No entanto, histórias são documentos resumidos, com apenas duas ou três sentenças, com as quais o representante dos clientes define o que ele deseja que o sistema faça, usando sua própria linguagem. Iremos aprofundar o estudo sobre histórias de usuários no Capítulo 3. Mas, por enquanto, gostaríamos de adiantar que as histórias são escritas em cartões de papel, normalmente a mão. Ou seja, em vez de documentos de requisitos detalhados, histórias são documentos simples, que focam nas funcionalidades do sistema, sempre na visão de seus usuários. Como exemplo, mostramos a seguir uma história de um sistema de perguntas e respostas — semelhante ao famoso Stack Overflow — que usaremos neste capítulo para explicar XP. Postar Pergunta Um usuário, quando logado no sistema, deve ser capaz de postar perguntas. Como é um site sobre programação, as perguntas podem incluir blocos de código, os quais devem ser apresentados com um leiaute diferenciado. Observe que a história tem um título (Postar Pergunta) e uma breve descrição, que não ocupa mais do que duas ou três sentenças. Costuma-se dizer que histórias são um lembrete para que depois esse requisito seja verbalmente detalhado pelo representante dos clientes. Depois de escritas pelo representante dos clientes, as histórias são estimadas pelos desenvolvedores. Ou seja, são os desenvolvedores que definem, mesmo que preliminarmente, quanto tempo será necessário para implementar as histórias escritas pelo representante dos clientes. Frequentemente, a duração de uma história é estimada em story points, em vez de horas ou homens/hora. Nesses casos, usa-se uma escala inteira para classificar histórias como possuindo um certo número de story points. O objetivo é definir uma ordem relativa entre as histórias. As histórias mais simples são estimadas como tendo tamanho igual a 1 story point; histórias que são cerca de duas vezes mais complexas do que as primeiras são estimadas como tendo 2 story points e assim por diante. Muitas vezes, usa-se uma sequência de Fibonacci para definir a escala de possíveis story points, como em 1, 2, 3, 5, 8, 13 story points. Nesse caso, o objetivo é criar uma escala que torne as tarefas progressivamente mais difíceis e, ao mesmo tempo, permita ao time realizar comparações similares à seguinte: será que o esforço para implementar essa tarefa que planejamos estimar com 8 story points é equivalente ao esforço de implementar uma tarefa na escala anterior (5 story points) e mais uma tarefa na próxima escala inferior (3 pontos)? Se sim, 8 story points é uma boa estimativa. Senão, o melhor é estimar a história com 5 story points. Aprofundamento: Uma técnica usada para estimar o tamanho de histórias é conhecida como Planning Poker. Ela funciona assim: o representante dos clientes seleciona uma história e a lê para os desenvolvedores. Após a leitura, os desenvolvedores interagem com o representante dos clientes para tirar possíveis dúvidas e conhecer melhor a história. Feito isso, cada desenvolvedor faz sua estimativa para o tamanho da história, de forma independente. Depois disso, eles ao mesmo tempo levantam cartões com a estimativa que pensaram, em story points. Esses cartões foram distribuídos antes e neles constam os números 1, 2, 3, 5, etc. Se houver consenso, o tamanho da história está estimado e passa-se para a próxima história. Senão, o time deve iniciar uma discussão, para esclarecer a razão das diferentes estimativas. Por exemplo, os desenvolvedores responsáveis pelas estimativas mais discrepantes podem explicar o motivo da sua proposta. Feito isso, realiza-se uma nova votação e o processo se repete, até que o consenso seja alcançado. A implementação das histórias ocorre em iterações, as quais têm uma duração fixa e bem definida, variando de uma a três semanas, por exemplo. As iterações, por sua vez, formam ciclos mais longos, chamados de releases, de dois a três meses, por exemplo. A velocidade de um time é o número de story points que ele consegue implementar em uma iteração. Sugere-se que o representante dos clientes escreva histórias que requeiram pelo menos uma release para serem implementadas. Ou seja, em XP, o horizonte de planejamento é uma release, isto é, alguns meses. Aviso: Em XP, a palavra release tem um sentido diferente daquele que se usa em gerência de configuração. Em gerência de configuração, uma release é uma versão de um sistema que será disponibilizada para seus usuários finais. Como já mencionamos em um aviso anterior, não necessariamente a versão do sistema ao final de uma release de XP precisa entrar em produção. Em resumo, para começar a usar XP precisamos de: Definir a duração de uma iteração. Definir o número de iterações de uma release. Um conjunto de histórias, escritas pelo representante dos clientes. Estimativas para cada história, feitas pelos desenvolvedores. Definir a velocidade do time, isto é, o número de story points que ele consegue implementar por iteração. Uma vez definidos os parâmetros e documentos acima, o representante do cliente deve priorizar as histórias. Para isso, ele deve definir quais histórias serão implementadas nas iterações da primeira release. Nessa priorização, deve-se respeitar a velocidade do time de desenvolvimento. Por exemplo, suponha que a velocidade de um time seja de 25 story points por iteração. Nesse caso, o representante do cliente não pode alocar histórias para uma iteração cujo somatório de story points ultrapasse esse limite. A tarefa de alocar histórias a iterações e releases é chamada de planejamento de releases (ou então planning game, que foi o nome adotado na primeira edição do livro de XP). Por exemplo, suponha o fórum de perguntas e respostas que mencionamos antes. A próxima tabela resume o resultado de um possível planejamento de releases. Nessa tabela, estamos assumindo que o representante dos clientes escreveu 8 histórias, que cada release possui duas iterações e que a velocidade do time é de 21 story points por iteração (veja que o somatório dos story points de cada iteração é exatamente igual a 21). História Story Points Iteração Release Cadastrar usuário 8 1 1 Postar perguntas 5 1 1 Postar respostas 3 1 1 Tela de abertura 5 1 1 Gamificar perguntas/respostas 5 2 1 Pesquisar perguntas/respostas 8 2 1 Adicionar tags 5 2 1 Comentar perguntas/respostas 3 2 1 A tabela anterior serve para reforçar dois pontos já mencionados: (1) as histórias em XP representam funcionalidades do sistema que se pretende construir; isto é, a implementação do sistema é dirigida por suas funcionalidades; (2) os desenvolvedores não opinam sobre a ordem de implementação das histórias; isso é decidido pelo representante dos clientes, que deve ser alguém capacitado e com autoridade para definir o que é mais urgente e importante para a empresa que está contratando o desenvolvimento do sistema. Uma vez realizado o planejamento de uma release, começam as iterações. Antes de mais nada, o time de desenvolvimento deve se reunir para realizar o planejamento da iteração. O objetivo desse planejamento é decompor as histórias de uma iteração em tarefas, as quais devem corresponder a atividades de programação que possam ser alocadas para um dos desenvolvedores do time. Por exemplo, a seguinte lista mostra as tarefas para a história Postar Perguntas, que é a primeira história que será implementada em nosso sistema de exemplo. Projetar e testar a interface Web, incluindo leiaute, CSS templates, etc. Instalar banco de dados, projetar e criar tabelas. Implementar a camada de acesso a dados. Instalar servidor e testar framework web. Implementar camada de controle, com operações para cadastrar, remover e atualizar perguntas. Implementar interface Web. Como regra geral, as tarefas não devem ser complexas, devendo ser possível concluí-las em alguns dias. Resumindo, um projeto XP é organizado em: releases, que são conjuntos de iterações, com duração total de alguns meses. iterações, que são conjuntos de tarefas resultantes da decomposição de histórias, com duração total de algumas semanas. tarefas, com duração de alguns dias. Definidas as tarefas, o time deve decidir qual desenvolvedor será responsável por cada uma. Feito isso, começa de fato a iteração, com a implementação das tarefas. Uma iteração termina quando todas as suas histórias estiverem implementadas e validadas pelo representante dos clientes. Assim, ao fim de uma iteração, as histórias devem ser mostradas para o representante dos clientes, que deve concordar que elas, de fato, atendem ao que ele especificou. XP defende ainda que os times, durante uma iteração, programem algumas folgas (slacks), que são tarefas que podem ser adiadas, caso necessário. Como exemplos, podemos citar o estudo de uma nova tecnologia, a realização de um curso online, preparar uma documentação ou manual ou mesmo desenvolver um projeto paralelo. Algumas empresas, como o Google, por exemplo, são famosas por permitir que seus desenvolvedores usem 20% de seu tempo para desenvolver um projeto pessoal (link). No caso de XP, folgas têm dois objetivos principais: (1) criar um buffer de segurança em uma iteração, que possa ser usado caso alguma tarefa demande mais tempo do que o previsto; (2) permitir que os desenvolvedores respirem um pouco, pois o ritmo de trabalho em projetos de desenvolvimento de software costuma ser intenso e desgastante. Logo, os desenvolvedores precisam de um tempo para realizarem algumas tarefas sem que exista uma cobrança imediata de resultados. Perguntas Frequentes Vamos agora responder algumas perguntas sobre as práticas de XP que acabamos de explicar. Qual a duração ideal de uma iteração? Difícil precisar, pois depende das características do time, da empresa contratante, da complexidade do sistema a ser desenvolvido, etc. Iterações curtas — por exemplo, de uma semana — propiciam feedback mais rápido. Porém, requerem um maior comprometimento dos clientes, pois toda semana um novo incremento de produto deve ser validado. Além disso, requerem que as histórias sejam mais simples. Por outro lado, iterações mais longas — por exemplo, de um mês — permitem que o time planeje e conclua as tarefas com mais tranquilidade. Porém, demora-se um pouco mais para receber feedback dos clientes. Esse feedback pode ser importante quando os requisitos são pouco claros. Por isso, uma escolha de compromisso seria algo como 2 ou 3 semanas. Outra alternativa recomendada consiste em experimentar, isto é, testar e avaliar diferentes durações, antes de decidir. O que o representante dos clientes faz durante as iterações? No início de uma release, cabe ao representante dos clientes escrever as histórias das iterações que farão parte dessa release. Depois, no final de cada iteração, cabe a ele validar e aprovar a implementação das histórias. Porém, durante as iterações, ele deve estar fisicamente disponível para tirar dúvidas do time. Veja que uma história é um documento muito resumido, logo é natural que surjam dúvidas durante a sua implementação. Por isso, o representante dos clientes deve estar sempre disponível para se reunir com os desenvolvedores, para tirar dúvidas e explicar detalhes relativos à implementação de histórias. Como escolher o representante dos clientes? Antes de mais nada, deve ser alguém que conheça o domínio do sistema e que tenha autoridade para priorizar histórias. Conforme detalhado a seguir, existem pelo menos três perfis de representante dos clientes: Suponha o desenvolvimento interno de um sistema, isto é, o departamento de sistemas da empresa X está desenvolvendo um sistema para um outro departamento Y, da mesma empresa. Nesse caso, o representante dos clientes deve ser um funcionário do departamento Y. Suponha que o time de desenvolvimento foi contratado para desenvolver um sistema para a empresa X. Ou seja, trata-se de um desenvolvimento terceirizado. Nesse caso, o representante do cliente deve ser um funcionário da empresa X, com pleno domínio da área do sistema e que vai ser um dos seus principais usuários, quando ele ficar pronto. Suponha que o time de desenvolvimento de uma empresa X foi designado para fazer um sistema para um público externo à empresa. Por exemplo, um sistema como aquele usado neste capítulo, similar ao Stack Overflow. Logo, os clientes do sistema não são funcionários de X, mas sim clientes externos. Nesse caso, o representante dos clientes deve ser alguém da área de marketing, vendas ou negócios da empresa X. Em última instância, pode ser o dono da empresa. Em qualquer caso, a sugestão é que seja uma pessoa próxima do problema e o mais distante possível da solução. Por isso mesmo, deve-se evitar que ele seja um desenvolvedor ou um gerente de projeto. O tipo de representante dos clientes que mencionamos neste item é, às vezes, chamado de um user proxy. Como definir a velocidade do time? Não existe bala de prata para essa questão. Essa definição depende da experiência do time e de seus membros. Se eles já participaram de projetos semelhantes àquele que estão iniciando, certamente essa deve ser uma questão menos difícil. Caso contrário, tem-se que experimentar e ir calibrando a velocidade nas iterações seguintes. Histórias podem incluir tarefas de instalação de infraestrutura de software? Não, histórias são especificadas pelo representante dos clientes, que é um profissional leigo em Engenharia de Software. Portanto, ele não costuma ter conhec

isto é, eles têm autonomia para decidir como e por quem as histórias serão implementadas. Ao lado do Backlog do Sprint, costuma-se anexar um quadro com tarefas a fazer, em andamento e finalizadas. Esse quadro — também chamado de Quadro Scrum (Scrum Board) — pode ser fixado nas paredes do ambiente de trabalho, permitindo que o time tenha diariamente uma sensação visual sobre o andamento do sprint. Veja um exemplo na próxima figura. Exemplo de Quadro Scrum, mostrando as histórias selecionadas para o sprint e as tarefas nas quais elas foram quebradas. Cada tarefa nesse quadro pode estar em um dos seguintes estados: a fazer, em andamento, em teste ou concluída. Uma decisão importante em projetos Scrum envolve os critérios para considerar uma história ou tarefa como concluídas (done). Esses critérios devem ser combinados com o time e ser do conhecimento de todos os seus membros. Por exemplo, em um projeto de desenvolvimento de software, para que uma história seja marcada como concluída pode-se exigir a implementação de testes de unidade, que devem estar todos passando, bem como a revisão do código por um outro membro do time. Além disso, o código deve ter sido integrado com sucesso no repositório do projeto. O objetivo desses critérios é evitar que os membros — de forma apressada e valendo-se de código de baixa qualidade — consigam mover suas tarefas para a coluna concluído. Um outro artefato comum em Scrum é o Gráfico de Burndown. A cada dia do sprint, esse gráfico mostra quantas horas são necessárias para se implementar as tarefas que ainda não estão concluídas. Isto é, no dia X do sprint ele informa que restam tarefas a implementar que somam Y horas. Logo, a curva de um gráfico de burndown deve ser declinante, atingindo o valor zero ao final do sprint, caso ele seja bem sucedido. Mostra-se a seguir um exemplo, assumindo-se um sprint com duração de 15 dias. Gráfico de Burndown, assumindo um sprint de 15 dias. Outros Eventos Vamos agora descrever mais três eventos Scrum, especificamente Reuniões Diárias, Revisão do Sprint e Retrospectiva. Scrum propõe que sejam realizadas Reuniões Diárias, de cerca de 15 minutos, das quais devem participar todos os membros do time. Essas reuniões para serem rápidas devem ocorrer com os membros em pé, daí serem também conhecidas como reuniões em pé (standup meetings, ou ainda daily scrum). Nelas, cada membro do time deve responder a três perguntas: (1) o que ele fez no dia anterior; (2) o que ele pretende fazer no dia corrente; (3) e se ele está enfrentando algum problema mais sério, isto é, um impedimento, na sua tarefa. Essas reuniões têm como objetivo melhorar a comunicação entre os membros do time, fazendo com que eles compartilhem e socializem o andamento do projeto. Por exemplo, dois desenvolvedores podem tomar ciência, durante a reunião diária, que eles vão começar a modificar o mesmo trecho de código. Portanto, seria recomendável que eles se reunissem, separadamente do resto do time, para discutir essas modificações. E, com isso, minimizar as chances de possíveis conflitos de integração. A Revisão do Sprint (Sprint Review) é uma reunião para mostrar os resultados de um sprint. Dela devem participar todos os membros do time e idealmente outros stakeholders, convidados pelo Dono do Produto, que estejam envolvidos com o resultado do sprint. Durante essa reunião o time demonstra, ao vivo, o produto para os clientes. Como resultado, todas as histórias do sprint podem ser aprovadas pelo Dono do Produto. Por outro lado, caso ele detecte problema em alguma história, ela deve voltar para o Backlog do Produto, para ser retrabalhada em um próximo sprint. O mesmo deve ocorrer com as histórias que o time não concluiu durante o sprint. A Retrospectiva é a última atividade de um sprint. Trata-se de uma reunião do time Scrum, com o objetivo de refletir sobre o sprint que está terminando e, se possível, identificar pontos de melhorias no processo, nas pessoas, nos relacionamentos e nas ferramentas usadas. Apenas para dar um exemplo, como resultado de uma retrospectiva, o time pode acordar sobre a importância de todos estarem presentes, pontualmente, nas reuniões diárias, pois nos últimos sprints alguns membros estão se atrasando. Veja, portanto, que uma retrospectiva não é uma reunião para lavar a roupa suja e para membros ficarem discutindo entre si. Se for necessário, isso deve ser feito em particular, em outras reuniões ou com a presença de gerentes da organização. Depois da retrospectiva, o ciclo se repete, com um novo sprint. Uma característica marcante de todos os eventos Scrum é terem uma duração bem definida, que é chamada de time-box da atividade. Por isso, esse termo aparece sempre em documentos Scrum. Por exemplo, veja essa frase do Scrum Guide oficial: o coração do método Scrum é um sprint, que tem um time-box de um mês ou menos e durante o qual um produto done, usável e que potencialmente pode ser colocado em produção é criado (link). O objetivo da fixação de time boxes é criar um fluxo contínuo de trabalho, bem como fomentar o compromisso da equipe com o sucesso do sprint e evitar a perda de foco. A próxima tabela mostra o time-box dos eventos Scrum. No caso de eventos com um time-box máximo (exemplo: planejamento do sprint), o valor recomendado refere-se a um sprint de um mês. Se o sprint for menor, o time-box sugerido deve ser também menor. Evento Time-box Planejamento do Sprint máximo de 8 horas Sprint menos de 1 mês Reunião Diária 15 minutos Revisão do Sprint máximo de 4 horas Retrospectiva máximo de 3 horas Exemplo: Escrita de um Livro Este livro está sendo escrito usando artefatos e eventos de Scrum. Claro que apenas alguns, pois o livro tem um único autor que, em certa medida, desempenha todos os papéis previstos por Scrum. Logo no início do projeto, os capítulos do livro foram planejados, constituindo assim o Backlog do Produto. A escrita de cada capítulo é considerada como sendo um sprint. Na reunião de Planejamento do Sprint, define-se a divisão do capítulo em seções, que são equivalentes às tarefas. Então começa-se a escrita de cada capítulo, isto é, tem início um sprint. Via de regra, os sprints são planejados para ter uma duração de dois meses. Para ficar mais claro, mostra-se a seguir o backlog do sprint atual, bem como o estado de cada tarefa, exatamente no momento em que se está escrevendo este parágrafo: História A fazer Em andamento Concluídas Cap. 2 - Processos de Desenvolvimento Kanban Quando não usar Métodos Ágeis Outros Processos Exercícios Scrum Introdução Manifesto Ágil XP Decidiu-se adotar um método ágil para escrita do livro para minimizar os riscos de desenvolver um produto que não atende às necessidades de nossos clientes, que, nesta primeira versão, são estudantes e professores brasileiros de disciplinas de Engenharia de Software, principalmente em nível de graduação. Assim, ao final de cada sprint, um capítulo é lançado e divulgado publicamente, de forma a receber feedback. Com isso, evita-se uma solução Waterfall, por meio da qual o livro seria escrito por cerca de dois anos, sem receber qualquer feedback. Para finalizar, vamos comentar sobre o critério para conclusão de um capítulo, ou seja, para definir que um capítulo está finalizado (done). Esse critério requer a leitura e revisão completa do capítulo, pelo autor do livro. Concluída essa revisão, o capítulo é divulgado preliminarmente para os membros do Grupo de Pesquisa em Engenharia de Software Aplicada, do DCC/UFMG. Perguntas Frequentes Antes de concluir a seção, vamos responder algumas perguntas sobre Scrum: O que significa a palavra Scrum? O nome não é uma sigla, mas uma referência à reunião de jogadores realizada em uma partida de rugby para decidir quem vai ficar com a bola após uma infração involuntária. O que é um squad? Esse termo é um sinônimo para time ágil ou time Scrum. O nome foi popularizado pela Spotify. Assim como times Scrum, squads são pequenos, cross-funcionais e auto-organizáveis. É comum ainda usar o nome tribo para denotar um conjunto de squads. O Dono do Produto pode ser um comitê? Em outras palavras, pode existir mais de um Dono de Produto em um time Scrum? A resposta é não. Apenas um membro do time exerce essa função. O objetivo é evitar decisões por comitê, que tendem a gerar produtos lotados de funcionalidades, mas que foram implementadas apenas para atender a determinados membros do comitê. Porém, nada impede que o Dono do Produto faça a ponte entre o time e outros usuários com amplo domínio da área do produto que está sendo construído. Na verdade, essa é uma tarefa esperada de Donos de Produto, pois às vezes existem requisitos que são do domínio de apenas alguns colaboradores da organização. Cabe então ao Dono do Produto intermediar as conversas entre os desenvolvedores e tais usuários. O Scrum Master deve exercer seu papel em tempo integral? Idealmente, sim. Porém, em times maduros, que conhecem e praticam Scrum há bastante tempo, às vezes não é necessário ter um Scrum Master em tempo integral. Nesses casos, existem duas possibilidades: (1) permitir que o Scrum Master desempenhe esse papel em mais de um time Scrum; (2) alocar a responsabilidade de Scrum Master a um dos membros do time. No entanto, caso a segunda alternativa seja adotada, o Scrum Master não deve ser também o Dono do Produto. A razão é que uma das responsabilidades do Scrum Master é exatamente acompanhar e auxiliar o Dono do Produto em suas tarefas de escrever e priorizar histórias do usuário. O Scrum Master precisa ter um diploma de nível superior em um curso da área de Computação? Não, pois sua função envolve remover impedimentos e assegurar que o time esteja seguindo os princípios de Scrum. Portanto, ele não é um resolvedor de problemas técnicos, tais como bugs, uso correto de frameworks, implementação de funcionalidades, etc. Por outro lado, isso não impede que um desenvolvedor técnico, com um curso superior na área de Computação, assuma as funções de Scrum Master, como vimos na resposta anterior. Existem também certificações para Scrum Master, as quais podem ser requeridas por empresas que decidem adotar Scrum. Além de histórias, quais outros itens podem fazer parte do Backlog do Produto? Também podem ser cadastrados no Backlog do Produto itens como bugs — principalmente aqueles mais complexos e que demandam dias para serem resolvidos — e também melhorias em histórias já implementadas. Existem gerentes quando se usa Scrum? A resposta é sim! De fato, times Scrum são autônomos para implementar as histórias priorizadas pelo Dono do Produto. Porém, um projeto demanda diversas outras decisões que devem ser tomadas em um nível gerencial. Dentre essas decisões, podemos citar as seguintes: Contratar e alocar membros para os times Scrum; ou seja, os desenvolvedores não têm autonomia para escolher em quais times eles vão trabalhar. Essa é uma decisão de mais alto nível e, portanto, tomada por gerentes. Decidir os objetivos e responsabilidades de cada time, incluindo o sistema — ou parte de um sistema — que o time irá desenvolver usando Scrum. Por exemplo, um time não decide, por conta própria, que a organização precisa de um novo sistema de contabilidade e então começa a desenvolvê-lo. Essa é uma decisão estratégica, que cabe aos gerentes e executivos da organização. Gerenciar e administrar questões de recursos humanos, incluindo contratações de novos funcionários, desligamentos de funcionários, promoções, transferências, treinamentos, etc. Avaliar se os resultados produzidos pelos times Scrum estão, de fato, gerando benefícios e valor para a organização. Kanban A palavra japonesa kanban significa cartão visual ou cartão de sinalização. Desde a década de 50, o nome também é usado para denotar o processo de produção just-in-time usado em fábricas japonesas, principalmente naquelas da Toyota, onde ele foi usado pela primeira vez. O processo também é conhecido como Sistema de Produção da Toyota (TPS) ou, mais recentemente, por manufatura lean. Em uma linha de montagem, os cartões são usados para controlar o fluxo de produção. No caso de desenvolvimento de software, Kanban foi usado pela primeira vez na Microsoft, em 2004, como parte de um esforço liderado por David Anderson, então um funcionário da empresa (link). Segundo Anderson, Kanban é um método que ajuda times de desenvolvimento a trabalhar em ritmo sustentável, eliminando desperdício, entregando valor com frequência e fomentando uma cultura de melhorias contínuas. Para começar a explicar Kanban, vamos usar uma comparação com Scrum. Primeiro, Kanban é mais simples do que Scrum, pois não usa nenhum dos eventos de Scrum, incluindo sprints. Também, não existe nenhum dos papéis (Dono do Produto, Scrum Master, etc.), pelo menos da forma rígida preconizada por Scrum. Por fim, não existe nenhum dos artefatos Scrum, com uma única e central exceção: o quadro de tarefas, que é chamado de Quadro Kanban (Kanban Board), e que inclui também o Backlog do Produto. O Quadro Kanban é dividido em colunas, da seguinte forma: A primeira coluna é o backlog do produto. Como em Scrum, usuários escrevem as histórias, que vão para o Backlog. As demais colunas são os passos que devem ser seguidos para transformar uma história do usuário em uma funcionalidade executável. Por exemplo, pode-se ter colunas como Especificação, Implementação e Revisão de Código. A ideia, portanto, é que as histórias sejam processadas passo a passo, da esquerda para a direita, como em uma linha de montagem. Além disso, cada coluna é dividida em duas subcolunas: em execução e concluídas. Por exemplo, a coluna implementação tem duas subcolunas: tarefas em implementação e tarefas implementadas. As tarefas concluídas em um passo estão aguardando serem puxadas, por um membro do time, para o próximo passo. Por isso, Kanban é chamado de um sistema pull. Mostra-se abaixo um exemplo de Quadro Kanban. Observe que existe uma história no backlog (H3), além de uma história (H2) que já foi puxada por algum membro do time para o passo de especificação. Existem ainda quatro tarefas (T6 a T9) que foram especificadas a partir de uma história anterior. Continuando, existem duas tarefas em implementação (T4 e T5) e existe uma tarefa implementada e aguardando ser puxada para revisão de código (T3). No último passo, existe uma tarefa em revisão (T2) e uma tarefa cujo processamento está concluído (T1). Por enquanto, não se preocupe com a sigla WIP que aparece em todos os passos, exceto no backlog. Vamos explicá-la em breve. Além disso, representamos as histórias e tarefas pelas letras H e T, respectivamente. Porém, em um quadro real, ambas são cartões auto-adesivos com uma pequena descrição. O Quadro Kanban pode ser assim montado em uma das paredes do ambiente de trabalho do time. Agora, mostraremos uma evolução do projeto. Isto é, alguns dias depois, o Quadro Kanban passou para o seguinte estado (as tarefas que avançaram no quadro estão sublinhadas e em vermelho). Veja que a história H2 desapareceu, pois foi quebrada em três tarefas (T10, T11 e T12). O objetivo da fase de especificação é exatamente transformar uma história em uma lista de tarefas. Continuando, T6 e T7 — que antes estavam esperando — entraram em implementação. Já T3 entrou na fase de revisão de código. Por fim, a revisão de T2 terminou. Veja ainda que, neste momento, não existe nenhuma tarefa implementada e aguardando ser puxada para revisão. Como em outros métodos ágeis, times Kanban são auto-organizáveis. Isso significa que eles têm autonomia para definir qual tarefa vai ser puxada para o próximo passo. Eles também são cross-funcionais, isto é, devem incluir membros capazes de realizar todos os passos do Quadro Kanban. Por fim, resta explicar o conceito de Limites WIP (Work in Progress). Via de regra, métodos de gerenciamento de projetos têm como objetivo garantir um ritmo sustentável de trabalho. Para isso, deve-se evitar duas situações extremas: (1) o time ficar ocioso boa parte do tempo, sem tarefa para realizar; ou (2) o time ficar sobrecarregado de trabalho e, por isso, não conseguir produzir software de qualidade. Para evitar a segunda situação — sobrecarga de trabalho — Kanban propõe um limite máximo de tarefas que podem estar em cada um dos passos de um Quadro Kanban. Esse limite é conhecido pelo nome Limite WIP, isto é, trata-se do limite máximo de cartões presentes em cada passo, contando aqueles na primeira coluna (em andamento) e aqueles na segunda coluna (concluídos) do passo. A exceção é o último passo, no qual o WIP aplica-se apenas à primeira subcoluna, já que não faz sentido aplicar um limite ao número de tarefas concluídas pelo time de desenvolvimento. A seguir, reproduzimos o último Quadro Kanban, mas com os limites WIP. Eles são os números que aparecem abaixo do nome de cada passo, na primeira linha do quadro. Ou seja, nesse Quadro Kanban, admite-se um máximo de 2 histórias em especificação; 5 tarefas em implementação; e 3 tarefas em revisão. Vamos deixar para explicar o limite WIP do passo Especificação por último. Mas podemos ver que existem 4 tarefas em implementação (T4, T5, T6 e T7). Portanto, abaixo do WIP desse passo, que é igual a 5 tarefas. Em Revisão de Código, o limite é de 3 tarefas e também está sendo respeitado, pois existe apenas uma tarefa em revisão (T3). Veja que para fins de verificar o limite WIP contam-se as tarefas em andamento (1a subcoluna de cada passo) e concluídas (2a subcoluna de cada passo), com exceção do último passo, no qual consideram-se apenas as tarefas da primeira subcoluna (T3, no exemplo). Agora vamos explicar o WIP do passo Especificação. Para verificar o WIP desse passo, deve-se somar as histórias em especificação (zero no quadro anterior) e as histórias que já foram especificadas. No caso, temos duas histórias especificadas. Ou seja, T8 e T9 são tarefas que resultaram da especificação de uma mesma história. E as tarefas T10, T11 e T12 são resultado da especificação de uma segunda história. Portanto, para fins do cálculo de WIP, temos duas histórias no passo, o que está dentro do seu limite, que também é 2. Para facilitar a visualização, costuma-se representar as tarefas resultantes da especificação de uma mesma história em uma única linha. Seguindo esse padrão, para calcular o WIP do passo Especificação, deve-se somar as histórias da primeira subcoluna (zero, no nosso exemplo) com o número de linhas na segunda coluna (duas). Ainda no quadro anterior, e considerando os limites WIP, tem-se que: A história H3, que está no backlog, não pode ser puxada para o passo Especificação, pois o WIP desse passo está no limite. Uma das tarefas já especificadas (T8 a T12) pode ser puxada para implementação, pois o WIP do passo está em 4, enquanto o limite é 5. Uma ou mais tarefas em implementação (T4 a T7) podem ser finalizadas, o que não altera o WIP do passo. A revisão de T3 pode ser finalizada. Apenas reforçando, o objetivo dos limites WIP é evitar que os times Kanban fiquem sobrecarregados de trabalho. Quando um desenvolvedor tem muitas tarefas para realizar — porque os limites WIP não estão sendo respeitados — a tendência é que ele não consiga concluir nenhuma dessas tarefas com qualidade. Como usual em qualquer atividade humana, quando assumimos muitos compromissos, a qualidade de nossas entregas cai muito. Kanban reconhece esse problema e, para que ele não ocorra, cria uma trava automática para impedir que os times aceitem trabalhos além da sua capacidade de entrega. Essas travas, que são os limites WIP, servem para uso interno do time e, mais importante ainda, para uso externo. Ou seja, elas são o instrumento de que um time dispõe para recusar trabalho extra que está sendo empurrado de cima para baixo por gerentes da organização, por exemplo. Calculando os Limites WIP Resta-nos agora explicar como os limites WIP são definidos. Existe mais de uma alternativa, mas vamos adotar uma adaptação de um algoritmo proposto por Eric Brechner — um engenheiro da Microsoft — em seu livro sobre o uso de Kanban no desenvolvimento de software (link). O algoritmo é descrito a seguir. Primeiro, temos que estimar quanto tempo em média uma tarefa vai ficar em cada passo do Quadro Kanban. Esse tempo é chamado de lead time (LT). No nosso exemplo, vamos supor os seguintes valores: LT(especificação) = 5 dias LT(implementação) = 12 dias LT(revisão) = 6 dias Veja que essa estimativa considera uma tarefa média, pois sabemos que vão existir tarefas mais complexas e mais simples. Veja ainda que o lead time inclui o tempo em fila, isto é, o tempo que a tarefa vai ficar na 2a subcoluna dos passos do Quadro Kanban aguardando ser puxada para o passo seguinte. Em seguida, deve-se estimar o throughput (TP) do passo com maior lead time do Quadro Kanban, isto é, o número de tarefas produzidas por dia nesse passo. No nosso exemplo, e na maioria dos projetos de desenvolvimento de software, esse passo é a Implementação. Assim, suponha que o time seja capaz de sustentar a implementação de 8 tarefas por mês. O throughput desse passo é então: 8 / 21 = 0.38 tarefas/dia. Veja que consideramos que um mês tem 21 dias úteis. Por fim, o WIP de cada passo é assim definido: WIP(passo) = TP * LT(passo). onde throughput refere-se ao throughput do passo mais lento, conforme calculado no item anterior. Logo, teremos os seguintes resultados: WIP(especificação) = 0.38 * 5 = 1.9 WIP(implementação) = 0.38 * 12 = 4.57 WIP(revisão) = 0.38 * 6 = 2.29 Arredondando para cima, os resultados finais ficam assim: WIP(especificação) = 2 WIP(implementação) = 5 WIP(revisão) = 3 No algoritmo proposto por Eric Brechner, sugere-se ainda adicionar uma margem de erro de 50% nos WIPs calculados, para acomodar variações no tamanho das tarefas, tarefas bloqueadas devido a fatores externos, etc. No entanto, como nosso exemplo é ilustrativo, não vamos ajustar os WIPs que foram calculados acima. Conforme afirmado, limites WIPs são o recurso oferecido por Kanban para garantir um ritmo de trabalho sustentável e a entrega de sistemas de software com qualidade. O papel desses limites é contribuir para que os desenvolvedores não fiquem sobrecarregados de tarefas e, consequentemente, propensos a baixar a qualidade de seu trabalho. Na verdade, todo método de desenvolvimento de software tende a oferecer esse tipo de recurso. Por exemplo, em Scrum existe o conceito de sprints com time-boxes definidos, cujo objetivo é evitar que times aceitem trabalhar em histórias que ultrapassam a sua velocidade de entrega. Adicionalmente, uma vez iniciado, o objetivo de um sprint não pode ser alterado, de forma a blindar o time de mudanças diárias de prioridade. No caso de métodos Waterfall, o recurso para garantir um fluxo de trabalho sustentável e de qualidade é a existência de uma fase detalhada de especificação de requisitos. Com essa fase, a intenção era oferecer aos desenvolvedores uma ideia clara do sistema que eles deveriam implementar. Lei de Little O procedimento para cálculo de WIPs explicado anteriormente é uma aplicação direta da Lei de Little, um dos resultados mais importantes da Teoria de Filas (link). A Lei de Little diz que o número de itens em um sistema de filas é igual à taxa de chegada desses itens multiplicado pelo tempo que cada item fica no sistema. Traduzindo para o nosso contexto, o sistema é um passo de um processo Kanban e os itens são tarefas. Assim, temos que: WIP: número de tarefas em um dado passo de um processo Kanban. Throughput (TP): taxa de chegada dessas tarefas nesse passo. Lead Time (LT): tempo que cada tarefa fica nesse passo. Ou seja, de acordo com a Lei de Little: WIP = TP * LT. Visualmente, podemos representar a Lei de Little como mostrado na próxima figura. Lei de Little: WIP = TP * LT Perguntas Frequentes Antes de concluir, vamos responder algumas perguntas sobre Kanban: Quais são os papéis que existem em Kanban? Ao contrário de Scrum, Kanban não define uma lista fixa de papéis. Cabe ao time e à organização definir os papéis que existirão no processo de desenvolvimento, tais como Dono do Produto, Testadores, etc. Como as histórias dos usuários são priorizadas? Kanban é um método de desenvolvimento mais leve que Scrum e mesmo que XP. Um dos motivos é que ele não define critérios de priorização de histórias. Como respondido na pergunta anterior, não se define, por exemplo, que o time tem que possuir um Dono do Produto, responsável por essa priorização. Veja que essa é uma possibilidade, isto é, pode existir um Dono do Produto em times Kanban. Mas outras soluções também são possíveis, como priorização externa, por um gerente de produto. Times Kanban podem realizar eventos típicos de Scrum, como reuniões diárias, revisões e retrospectivas? Sim, apesar de Kanban não prescrever a realização desses eventos. Porém, também não há um veto explícito aos mesmos. Cabe ao time decidir quais eventos são importantes, quando eles devem ser realizados, com que duração, etc. Em vez de um Quadro Kanban físico, com adesivos em uma parede ou em um quadro branco, pode-se usar um software para gerenciamento de projetos? Kanban não proíbe o uso de software de gerenciamento de projetos. Porém, recomenda-se o uso de um quadro físico, pois um dos princípios mais importantes de Kanban é a visualização do trabalho pelo time, de forma que seus membros possam a qualquer momento tomar conhecimento do trabalho em andamento e de possíveis problemas e gargalos que estejam ocorrendo. Em alguns casos, chega-se a recomendar a adoção de ambas soluções: um quadro físico, mas com um backup em um software de gerenciamento de projetos, que possa ser acessado pelos gerentes e executivos da organização. Quando Não Usar Métodos Ágeis? Apesar de métodos ágeis — como aqueles estudados nas seções anteriores — terem alcançado um sucesso inquestionável, é bom lembrar que não existe bala de prata em Engenharia de Software, conforme comentamos no Capítulo 1. Assim, neste capítulo vamos discutir cenários e domínios nos quais práticas ágeis podem não ser adequadas. No entanto, a pergunta que abre essa seção não admite uma resposta simples, como, por exemplo, sistemas das áreas X, Y e Z não devem usar métodos ágeis; e os demais devem usar. Em outras palavras, sistemas de qualquer área podem se beneficiar de pelo menos algumas das práticas propostas por métodos ágeis. Por outro lado, existem práticas que não são recomendadas para determinados tipos de sistemas, organizações e contextos. Assim, vamos responder a pergunta proposta em uma granularidade mais fina. Isto é, vamos comentar a seguir sobre quando não usar determinadas práticas de desenvolvimento ágil. Design Incremental. Esse tipo de design faz sentido quando o time tem uma primeira visão do design do sistema. Se o time não tem essa visão, ou o domínio do sistema é novo e complexo, ou o custo de mudanças futuras é muito alto, recomenda-se adotar uma fase de design e análise inicial, antes de partir para iterações que requeiram implementações de funcionalidades. Histórias do Usuário. Histórias são um método leve para especificação de requisitos, que depois são clarificados com o envolvimento cotidiano de um representante dos clientes no projeto. Porém, em certos casos, pode ser importante ter uma especificação detalhada de requisitos no início do projeto, principalmente se ele for um projeto de uma área totalmente nova para o time de desenvolvedores. Envolvimento do Cliente. Se os requisitos do sistema são estáveis e de pleno conhecimento do time de desenvolvedores, não faz sentido ter um Representante dos Clientes ou Dono do Produto integrado ao time. Por exemplo, esse papel não é importante no desenvolvimento de um compilador para uma linguagem conhecida, com uma gramática e semântica consolidadas. Documentação Leve e Simplificada. Em certos domínios, documentações detalhadas de requisitos e de projeto são mandatórias. Por exemplo, sistemas cujas falhas podem causar a morte de seres humanos, como aqueles das áreas médicas e de transporte, costumam demandar certificação por uma entidade externa, que pode exigir uma documentação detalhada, além do código fonte. Times Auto-organizáveis. Times ágeis são autônomos e empoderados para trabalhar sem interferências durante o time-box de uma iteração. Consequentemente, eles não precisam prestar contas diárias para os gerentes e executivos da organização. No entanto, essa característica pode ser incompatível com os valores e cultura de certas organizações, principalmente aquelas com uma tradição de níveis hierárquicos e de controle rígidos. Contratos com Escopo Aberto. Em contratos com escopo aberto, a remuneração é por hora trabalhada. Assim, a empresa que assina o contrato não tem — no momento da assinatura — uma ideia precisa de quais funcionalidades serão implementadas e nem do prazo e custo do sistema. Algumas organizações podem não se sentir seguras para assinar esse tipo de contrato, principalmente quando elas não têm uma experiência prévia com desenvolvimento ágil ou referências confiáveis sobre a empresa contratada. Para concluir, é importante mencionar que duas práticas ágeis são atualmente adotadas na grande maioria de projetos de software: Times pequenos, pois o esforço de sincronização cresce muito quando os times são compostos por dezenas de membros. Iterações (ou sprints), mesmo que com duração maior do que aquela típica de métodos ágeis. Por exemplo, iterações com duração de dois ou três meses, em vez de iterações com menos de 30 dias. Na verdade, entre o surgimento de Waterfall e de métodos ágeis, alguns métodos iterativos foram propostos, isto é, métodos com pontos de validação ao longo do desenvolvimento. Na próxima seção, iremos estudar dois desses métodos. Outros Métodos Iterativos A transição entre Waterfall — dominante nas décadas de 70 e 80 — e métodos ágeis — que começaram a surgir na década de 90, mas que só ganharam popularidade no final dos anos 2000 — foi gradativa. Como em métodos ágeis, esses métodos surgidos nesse período de transição possuem o conceito de iterações. Ou seja, eles não são estritamente sequenciais, como em Waterfall. Porém, as iterações têm duração maior do que aquela usual em desenvolvimento ágil. Em vez de poucas semanas, elas duram alguns meses. Por outro lado, eles preservam características relevantes de Waterfall, como ênfase em documentação e em uma fase inicial de levantamento de requisitos e depois de design. Um exemplo de proposta de processo surgida nessa época é o Modelo em Espiral, proposto por Barry Boehm, em 1986 (link). Nesse modelo, um sistema é desenvolvido na forma de uma espiral de iterações. Cada iteração, ou volta completa na espiral, inclui quatro etapas (acompanhe também na próxima figura): Definição de objetivos e restrições, tais como custos, cronogramas, etc. Avaliação de alternativas e análise de riscos. Por exemplo, pode-se chegar à conclusão de que é mais interessante comprar um sistema pronto, do que desenvolver o sistema internamente. Desenvolvimento e testes, por exemplo, usando Waterfall. Ao final dessa etapa, deve-se gerar um protótipo que possa ser demonstrado aos usuários do sistema. Planejamento da próxima iteração ou então tomar a decisão de parar, pois o que já foi construído é suficiente para atender às necessidades da organização. Modelo Espiral. Cada iteração é dividida em quatro etapas. Assim, o Modelo em Espiral produz, a cada iteração, versões mais completas de um sistema, começando da versão gerada no centro da espiral. Porém, cada iteração, somando-se as quatro fases, pode levar de 6 a 24 meses. Portanto, mais do que em XP e Scrum. Uma outra característica importante é a existência de uma fase explícita de análise de riscos, da qual devem resultar medidas concretas para mitigar os riscos identificados no projeto. O Processo Unificado (UP), proposto no final da década de 90, é outro exemplo de método iterativo de desenvolvimento. UP foi proposto por profissionais ligados a uma empresa de consultoria e de ferramentas de apoio ao desenvolvimento de software chamada Rational, que em 2003 seria comprada pela IBM. Especificamente, a versão do método implementada pela Rational é chamada de Rational Unified Process (RUP). Devido a suas origens, o RUP é vinculado a duas tecnologias específicas: Linguagem de modelagem UML, pois muitos dos resultados do RUP são documentados e representados usando-se diagramas gráficos de UML. No Capítulo 4, iremos estudar UML com mais calma. Por enquanto, vamos ressaltar que a proposta era ter uma linguagem de modelagem unificada (UML) e também um processo unificado (RUP), ambos propostos pelo mesmo grupo de profissionais. Ferramentas de apoio ao projeto e análise de software, conhecidas como ferramentas CASE (Computer-Aided Software Engineering). O nome é uma analogia com ferramentas CAD (Computer Aided-Design), usadas em projetos de Engenharia Civil, Engenharia Mecânica, Arquitetura, etc. A ideia era que o projeto e análise de um sistema deveriam ser integralmente baseados em diagramas UML. Mas esses diagramas não seriam desenhados em papel e sim usando-se ferramentas computacionais (veja um exemplo na figura da próxima página). A Rational, além de propor o seu método de desenvolvimento, também vendia licenças de uso de ferramentas CASE. Projeto usando ferramenta CASE. Figura gentilmente cedida pelos desenvolvedores do sistema ArgoUML. O RUP propõe que o desenvolvimento seja decomposto nas seguintes fases: Inception (às vezes, traduzida como iniciação ou concepção): que inclui análise de viabilidade, definição de orçamentos, análise de riscos e definição de escopo do sistema. Ao final dessa fase, o caso de negócio (business case) do sistema deve estar bem claro. Pode-se inclusive decidir que não vale a pena desenvolver o sistema, mas sim comprar um sistema pronto. Elaboração: que inclui especificação de requisitos (via diagramas de casos de uso de UML, por exemplo), definição da arquitetura do sistema, bem como de um plano para o seu desenvolvimento. Ao final dessa fase, todos os riscos identificados na fase anterior devem estar devidamente controlados e mitigados. Construção: na qual se realiza o projeto de mais baixo nível, implementação e testes do sistema. Ao final dessa fase, deve ser disponibilizado um sistema funcional, incluindo documentação e manuais, que possam ser validados pelos usuários. Transição: na qual ocorre a disponibilização do sistema para produção, incluindo a definição de todas as rotinas de implantação, como políticas de backup, migração de dados de sistemas legados, treinamento da equipe de operação, etc. Assim como no Modelo Espiral, pode-se repetir várias vezes o processo; ou seja, o desenvolvimento é incremental, com novas funcionalidades sendo entregues a cada ciclo. Adicionalmente, pode-se repetir cada um das fases. Por exemplo, construção pode ser dividida em duas iterações, cada uma construindo uma parte do produto. A próxima figura ilustra o modelo de iterações do RUP. Fases e iterações do RUP. Iterações são possíveis em cada fase (auto-laços). E também pode-se repetir todo o ciclo (laço externo), para gerar mais um incremento de produto. O RUP define também um conjunto de disciplinas de engenharia que incluem por exemplo: modelagem de negócios, definição de requisitos, análise e design, implementação, testes e implantação. Essas disciplinas — ou fluxos de trabalho — podem ocorrer em qualquer fase. Porém, espera-se que algumas disciplinas sejam mais intensas em determinadas fases, como mostra a próxima figura. No projeto ilustrado, tarefas de modelagem de negócio estão concentradas nas fases iniciais do projeto (inception e elaboração) e quase não ocorrem nas fases seguintes. Por outro lado, implementação está concentrada na fase de Construção. Fases (na horizontal) e disciplinas (na vertical) de um projeto desenvolvido usando o RUP. A área da curva mostra a intensidade da disciplina durante cada fase (imagem da Wikipédia, licença: domínio público). Kent Beck, Cynthia Andres. Extreme Programming Explained: Embrace Change. Addison-Wesley, 2nd edition, 2004. Kent Beck. Embracing Change with Extreme Programming. IEEE Computer, vol. 32, issue 10, p. 70-77, 1999. Kent Beck, Martin Fowler. Planning Extreme Programming. Addison-Wesley, 2000. Ken Schwaber, Jeff Sutherland. The Scrum Guide, 2017. Kenneth Rubin. Essential Scrum: A Practical Guide to the Most Popular Agile Process. Addison-Wesley, 2012 Eric Brechner. Agile Project Management with Kanban. Microsoft Press, 2015. David Anderson. Kanban. Blue Hole Press, 2013. Ian Sommerville. Engenharia de Software. Pearson, 10a edição, 2019. Hans van Vliet. Software Engineering: Principles and Practice. 3rd edition. Wiley, 2008. Armando Fox, David Patterson. Construindo Software como Serviço: Uma Abordagem Ágil Usando Computação em Nuvem. Strawberry Canyon LLC. 1a edição, 2014. Exercícios de Fixação 1. Como XP preconiza que devem ser os contratos de desenvolvimento de software? 2. Quais as diferenças entre XP e Scrum? 3. Times Scrum são ditos cross-funcionais e auto-organizáveis. Por quê? Defina esses termos. 4. Em Scrum, qual a diferença entre as histórias do topo e do fundo do Backlog do Produto? 5. O que são e para que servem story points? 6. Em Scrum, qual a diferença entre uma sprint review e uma retrospectiva? 7. Um sprint pode ser cancelado? Se sim, por quem e por qual motivo? Para responder a essa questão, consulte antes o Scrum Guide (link), que é o guia que documenta a versão oficial de Scrum. 8. Procure pensar em um sistema de uma área da qual tenha algum conhecimento. (a) Escreva então uma história para esse sistema (veja que histórias são especificações resumidas de funcionalidades, com 2-3 sentenças). (b) Em seguida, quebre a história que definiu em algumas tarefas (de forma semelhante ao que fizemos no sistema similar ao Stack Overflow, usado como exemplo na seção sobre XP). (c) Existem dependências entre essas tarefas? Ou elas podem ser implementadas em qualquer ordem? 9. Suponha dois times, A e B, atuando em projetos diferentes, contratados por empresas distintas, sem conexões entre eles. Porém, ambos os times adotam sprints de 15 dias e ambos possuem 5 desenvolvedores. Nos seus projetos, o time A considera que sua velocidade é de 24 pontos. Já o time B assume uma velocidade de 16 pontos. Pode-se afirmar que A é 50% mais produtivo que B? Justifique sua resposta. 10. Quais são as principais diferenças entre Scrum e Kanban? 11. Quais são as diferenças entre um Quadro Scrum e um Quadro Kanban? 12. Qual o erro existe no seguinte Quadro Kanban? 13. Suponha o seguinte quadro Kanban. Neste momento, o time não consegue trabalhar na especificação de novas histórias, pois o WIP do passo Especificação está sendo totalmente preenchido por itens esperando movimentação para o passo seguinte (Implementação). O que seria mais recomendado neste momento: (a) desrespeitar o WIP e já puxar uma nova história do Backlog para Especificação; ou (b) ajudar o time nas três tarefas em Validação, de forma a desbloquear o fluxo do processo. 14. Por que se recomenda que os limites WIP calculados usando a Lei de Little sejam incrementados, por exemplo em 50%, de forma a admitir uma margem de erro? Em outros palavras, quais eventos podem originar esses erros na estimativa dos WIPs? 15. Descreva os principais recursos oferecidos por Waterfall, Scrum e Kanban para controlar riscos e garantir um fluxo de trabalho sustentável e que propicie o desenvolvimento de software com qualidade. 16. Seja um processo Kanban, dividido em quatro passos. A tabela a seguir informa o lead time de cada um deles e o throughout do passo C, que é o passo mais lento. Com base nesses valores, calcule o WIP de cada passo (última coluna da tabela). Passo Lead Time (médio, dias) Throughput (tarefas/dia) WIP A 4 - B 3 - C 10 0.5 D 5 - 17. Seja o seguinte gráfico, que mostra — para um determinado sistema — como os custos de mudanças variam conforme a fase do desenvolvimento em que elas são realizadas. (