UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

DEFINIÇÃO DE PROCESSOS REUTILIZÁVEIS PARA CONSTRUÇÃO ÁGIL DE SERVIÇOS EMPREGANDO XP E PRINCÍPIOS SOA

Felipe Carvalho

Orientadores

Leonardo Guerreiro Azevedo

Gleison dos Santos Souza

RIO DE JANEIRO, RJ - BRASIL
SETEMBRO de 2014

DEFINIÇÃO DE PROCESSOS REUTILIZÁVEIS PARA CONSTRUÇÃO ÁGIL DE SERVIÇOS EMPREGANDO XP E PRINCÍPIOS SOA

Felipe Abrantes Carvalho

DISSERTAÇÃO APRESENTADA COMO REQUISITO PARCIAL PARA OBTENÇÃO DO TÍTULO DE MESTRE PELO PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA DA UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO (UNIRIO). APROVADA PELA COMISSÃO EXAMINADORA ABAIXO ASSINADA.

Aprovada por:

Leonardo Guerreiro Azevedo, D.Sc., UNIRIO

Gleison dos Santos Souza, D.Sc., UNIRIO

Leonardo Gresta Paulino Murta, D.Sc., UFF

Flavia Maria Santoro, D.Sc., UNIRIO

Alexandre Luis Correa, D.Sc., UNIRIO

RIO DE JANEIRO, RJ - BRASIL
SETEMBRO de 2014

Carvalho, Felipe Abrantes.
C331 Definição de processos reutilizáveis para construção ágil de serviços empregando XP e princípios SOA / Felipe Abrantes. Carvalho, 2014.
365 f. ; 30 cm + CD-ROM.

Orientador: Leonardo Guerreiro Azevedo.
Coorientador: Coorientador: Gleison dos Santos Souza.
Dissertação (Mestrado em Informática) - Universidade Federal do Estado do Rio de Janeiro, Rio de Janeiro, 2014.

Arquitetura Orientada a Serviços (Computador). 2. eXtreme programming. 3. Informática - Serviços ao cliente. 4. Software - Desenvolvimentos. 5. Métodos ágeis. 6. Linha de processos de software. I. Azevedo, Leonardo Guerreiro. II. Souza, Gleison dos Santos. III. Universidade Federal do Estado do Rio de Janeiro. Centro de Ciências Exatas e Tecnológicas. Curso de Mestrado em Informática. IV. Título.

CDD - 005.1

Para os meus pais.

Agradecimentos

Aos meus pais, Henrique e Ivone, sempre, que, com sacrifício e luta, me deram condições de alcançar a condição de mestre – infelizmente, benefício para poucos em nosso país. E a Deus, por ter sempre colocado tantas pessoas boas em meu caminho.

CARVALHO, FELIPE. **Definição de Processos Reutilizáveis para Construção Ágil de Serviços empregando XP e Princípios SOA**. UNIRIO, 2014. 365 páginas. Dissertação de Mestrado. Departamento de Informática Aplicada, UNIRIO.

RESUMO

Nas últimas décadas, a economia global tem se reestruturado em direção a um modelo de negócios orientado a serviços, ou seja, as organizações tem se posicionado de acordo com as competências em que são especialistas e organizado uma rede de colaboração com parceiros, a fim de reutilizar soluções prontas para entregar uma solução a seus clientes. Desta forma, o *time-to-market* de uma solução diminui, ao passo que o retorno sobre o investimento aumenta, na medida em que os ativos organizacionais são reutilizados com maior frequência.

Em paralelo, a velocidade de mudanças no mercado também vem aumentando, trazendo consigo maior volatilidade de requisitos e demandando maior flexibilidade e agilidade na resposta a mudanças. Assim, face à necessidade de menor tempo de resposta, novas formas de trabalhar foram surgindo, propondo entregas parciais e maior adaptabilidade a requisitos voláteis.

O conceito de Arquitetura Orientada a Serviço surgiu nesse contexto, procurando oferecer maior flexibilidade e menor tempo de resposta às necessidades do mercado e de organizações. Pelo mesmo motivo foram formulados uma série de métodos voltados a um desenvolvimento de software mais leve e menos burocrático. Estes métodos foram posteriormente denominados "Métodos Ágeis".

No entanto, apesar de tais similaridades, não há consenso na literatura sobre a aplicabilidade desta combinação em um processo de desenvolvimento que entregue soluções orientadas a serviço. De fato, há carência de métodos estruturados e repetíveis que permitam implementar esta combinação de forma apropriada.

Esta dissertação tem por objetivo definir uma linha de processos de *software* voltada a construção de soluções orientadas a serviço por meio de práticas ágeis e que, além disso,

leve em consideração o contexto da organização onde está sendo instanciada. Desta forma, espera-se que organizações possam instanciar variantes desta linha de processos com facilidade e de forma desassistida, respeitando sua realidade, a realidade de seu nicho de mercado, as características de sua força de trabalho, seu ferramental, sua cultura, entre outros fatores.

Palavras-chave: Service-Oriented Architecture, eXtreme Programming, Métodos Ágeis, Linha de Processos de Software, Construção de Serviços.

ABSTRACT

In the last decades, global economy has reestructured towards a service-oriented business model, meaning organizations have been positioning according to their core competencies e organized a collaboration network with partners, as to reuse pre-made solutions to deliver a solution to their customers. This way, a given solution's time-to-market decreases, whereas return over investiment increases, as organizational assets are reused more often.

In parallel, the speed of changes in the market has been increasing as well, bringing more volatility to requirements and demanding greater flexibility and agility to respond to changes. Due to the need for a smaller response time, new ways of working have been uncovered, proposing incremental deliveries and greater adaptability to unstable requirements.

The concept of Service-Oriented Architecture has emerged within this context, proposing greater flexibility and smaller response time to the needs of markets and organizations. For the same reason, a series of lightweight and non-burocratic methods were proposed. Those methods were aftewards called "Agile Methods".

Despite such similarities, there's no consensus in literature about how applicable is this combination into a software development method that delivers service-oriented solutions. In fact, there's a lack of structured and repeatable methods that allow this combination to be properly implemented.

This dissertation aims to define a software process line which purpose is to build service-oriented solutions via agile practices, in a way that takes into consideration the context of the organization where it is being implemented. This way, it is expected that organizations my easily and unassistedly instantiate variations of this software process line, according to its reality, the reality of their market niche, their workforce, their tooling, their culture, among other factors.

Keywords: Service-Oriented Architecture, eXtreme Programming, Agile Methods, Software Process Line, Service Construction.

Sumário

Lista de Figuras

Lista de Tabelas

1. Introdução

1.1 Contexto acadêmico

O século XXI trouxe uma nova configuração à economia global: dinamismo, novos canais de comunicação, competição global, mercados desregulamentados. Formou-se então um cenário onde organizações se comunicam globalmente e negociam diretamente [1], estando, assim, inseridas em um contexto onde é necessário responder rapidamente a mudanças no mercado. De fato, as economias globais evoluíram para "economias baseadas em serviços" [2], onde os modelos de negócio de organizações tradicionalmente orientadas à produção estão se reposicionando e se adaptando a um modelo guiado por serviços.

As organizações vêm se reestruturando de acordo com as competências em que são especialistas [1]. Para poder centrar-se em tais competências, estas organizações precisam estabelecer uma rede de colaboração com parceiros estratégicos de modo a conseguir entregar um produto final aos seus clientes. De forma mais geral, as organizações se vêem na necessidade de encontrar formas mais ágeis de trabalhar. Neste contexto, o termo "agilidade" denota a habilidade de lidar com requisitos e ambientes instáveis de modo a aumentar suas vantagens competitivas [2].

O conceito de Arquitetura Orientada a Serviços (ou Service-Oriented Architecture - SOA) surgiu em resposta a estas necessidades organizacionais para tornar ágil o atendimento às mudanças no negócio. SOA busca alcançar menor tempo de resposta e agilidade através do reuso de ativos organizacionais, cooperação entre interessados e maximização do valor da organização [3] [4] [2] [5] [6].

De fato, face à necessidade de responder com mais agilidade às mudanças demandadas pelo mercado, as organizações perceberam que maneiras tradicionais de trabalho não conseguiam acompanhar tais mudanças [7]. Projetos baseados em requisitos congelados,

prazos longos e interessados distantes não têm mais condição de entregar produtos em um ambiente de requisitos instáveis. Existem diversas diferenças entre as engenharias de software tradicional e orientada a serviço [6] [8]. Por exemplo, a orientação a serviços não implementa um sistema como um bloco auto-contido; a perspectiva é diferente, a solução de um problema é dada pela composição de serviços de granularidade que atuam como blocos de construção para formar um sistema orientado a serviço [8]. Outra diferença se refere a arquitetura da solução. Soluções orientadas a serviço são, por definição, abertas, o que significa dizer que sua arquitetura pode ser modificada em tempo de execução, mesmo depois de sua instalação em ambiente de produção. Quanto ao ambiente de execução, soluções SOA assumem que não haverá um ambiente de execução estável, a incerteza é inerente ao projeto de soluções orientadas a serviço. Por fim, uma outra diferença diz respeito a requisitos não-funcionais. Um sistema "tradicional" é projetado para lidar com um conjunto finito e bem definido de requisitos não-funcionais. Soluções orientadas a serviço, por outro lado, pressupõem potencial de reuso por consumidores diversos, com necessidades distintas. Portanto, novos consumidores podem passar a consumir a solução a qualquer momento, podendo trazer novas necessidades não-funcionais.

No início da década de 2000, foram publicados diversos artigos que destacavam os problemas inerentes ao desenvolvimento de software guiado por documentação excessiva e processos pesados. Nestes trabalhos, era defendido o uso de métodos mais leves, voltados para a interação direta com o cliente e resposta mais rápida às mudanças no negócio. Em 2001, foi realizado encontro entre estes autores para discutir maneiras melhores de se produzir software e, como resultado, o Manifesto Ágil foi publicado [9]. Neste manifesto, são valorizados colaboração, capacidade de resposta a mudanças, entendimento do negócio, simplicidade e agilidade. Métodos propostos atendendo a estas características passaram a ser denominados "métodos ágeis".

Assim, percebe-se que os conceitos de métodos ágeis e SOA surgiram a partir de preocupações semelhantes, tais como: flexibilidade, resposta rápida para mudanças, entendimento do negócio. No entanto, apesar desta similaridade, não há consenso na literatura sobre a validade de se combinar ambos. Segundo KROGDAHL, LUEF & STEINDL [10], há pesquisadores que afirmam que esta combinação é equivalente a combinar "óleo e água". Um exemplo desta discordância é descrito a seguir. Num contexto de SOA, o software é construído por meio da composição de serviços fracamente acoplados, com suas interfaces expostas e comunicação estabelecida por meio de mensagens [11]. Uma vez que o serviço é publicado, se torna difícil modificar sua interface, devido ao potencial impacto sobre múltiplos consumidores. Logo, torna-se necessário projetar serviços com bastante cuidado, de forma a aumentar a flexibilidade de suas interfaces, fomentar seu

reuso e minimizar a necessidade de modificações. Por outro lado, segundo FOWLER [12], prever reuso não é uma tarefa trivial, independente do contexto da aplicação, tornando-se ainda mais difícil quando fronteiras organizacionais são cruzadas. Logo, o princípio de projeto de interfaces visando máximo reuso e flexibilidade não segue o princípio de projeto evolutivo, no qual se baseiam os métodos ágeis.

Por outro lado, LANKHORST [2] afirma que é possível combinar métodos ágeis e desenvolvimento de serviços. Ele enfatiza que a incerteza é inerente ao desenvolvimento de serviços, tratando-se de um "problema perverso", que abrange aspectos de complexidade social, onde os métodos ágeis e suas práticas de envolvimento do cliente, *feedback* contínuo, desenvolvimento incremental, entre outros, são mecanismos para lidar esta complexidade e dinamismo.

Alheios a discussão sobre a aplicabilidade da combinação de métodos ágeis a SOA, diversos autores relatam dificuldades inerentes ao desenvolvimento de serviços, a serem detalhadas na Seção 1.2.

1.2 Motivação

Dado o contexto industrial, o objetivo inicial da pesquisa era encontrar um método que possibilitasse a entrega de soluções orientadas a serviço por meio de práticas ágeis. De forma geral, o objetivo da pesquisa originalmente era responder a questão de pesquisa: "como combinar métodos ágeis e SOA?".

Num espectro mais amplo, percebe-se que diversos autores relatam problemas vivenciados por equipes trabalhando em soluções orientadas a serviço. TAN *et al.*, em sua proposta do método WSIM-XP [14], relata alguns dos problemas que motivaram sua pesquisa:

- (i) Soluções orientadas a serviço envolvem times multi-disciplinares formados por programadores dos provedores de serviço, programadores de interface, analistas de testes, gerentes de projeto, usuários, entre outros. Os membros destes times não podem trabalhar isoladamente, pois as decisões tomadas por um afetam todos os outros. Há interseção e sobreposição de papéis, o que torna impossível estabelecer de forma irrestrita a separação de responsabilidades.
- (ii) Serviços demandam testes que levem em consideração a interoperabilidade entre diversos sistemas, necessitando, com frequência, rápida disponibilização e integração com clientes e parceiros de negócio.

- (iii) O desenvolvimento de serviços requer a participação ativa do cliente, priorizando suas necessidades, definindo o domínio do problema e tomando decisões, uma vez que este detém o maior conhecimento do processo que o software tentará reproduzir.

KARSTEN & CANNIZZO [15] também relatam alguns obstáculos que tiveram que ser superados por equipes trabalhando em produtos orientados a serviço:

- (iv) O crescimento das equipes dificultou manter a coesão da visão do produto final, bem como manter todos atualizados sobre o progresso dos times envolvidos.
- (v) A coordenação do trabalho de times distribuídos é, no melhor caso, difícil durante a maior parte do tempo.
- (vi) Comunicação em alta velocidade é um dos aspectos mais críticos do desenvolvimento de software, especialmente no caso de grupos distribuídos geograficamente.

GU & LAGO [6] relatam outras características inerentes ao desenvolvimento de serviços:

- (vii) Construção de software orientada pelo reuso.
- (viii) Maior envolvimento dos interessados no produto.
- (ix) Necessidade de um entendimento profundo do modelo de negócios por todos os envolvidos.
- (x) Distribuição de serviços através de fronteiras organizacionais.
- (xi) Maior alinhamento da TI ao negócio.

Por fim, ERL [4] também explicita alguns pontos que, por natureza, aumentam a complexidade de soluções orientadas a serviço:

- (xii) Dada a sua constante ênfase no reuso, grande parte do repositório de serviços de uma organização é composta de serviços agnósticos, com potencial de atender a requisitos de múltiplos consumidores. Com isso, apesar da possibilidade de se estabelecer uma arquitetura extremamente normalizada e simples, há também a possibilidade de se aumentar a complexidade do projeto dos serviços, bem como da arquitetura como um todo.

- (xiii) A estratégia de preferencia para a entrega de serviços é começar pelo mapeamento de todos os serviços planejados e seus relacionamentos, de forma a demandar menor esforço de modificações posteriores a estes serviços. Porém, ao mesmo tempo, esta estratégia impõe um grande esforço de análise inicial, a qual, muitas vezes, é inviável.

Verificou-se que os autores referenciavam com mais frequência atividades referentes ao planejamento e controle de projetos ágeis, do que atividades relativas a efetiva construção de software, conforme visto em CARVALHO & AZEVEDO [17]. Havia poucas respostas a questões relativas a melhores práticas de construção de serviços, como, por exemplo, como mitigar o impacto de requisitos instáveis durante o desenvolvimento de uma solução orientada a serviço; ou como favorecer a interoperabilidade entre diferentes plataformas consumidoras de serviços na distribuição destes através de fronteiras organizacionais. Por esta razão, de forma a oferecer uma forma de se construir soluções orientadas a serviço tão robustas quanto flexíveis, o foco deste trabalho se volta à fase de construção de um ciclo de desenvolvimento de *software*.

Dado o foco do trabalho voltado à fase de construção, o passo seguinte seria verificar se o uso de métodos ágeis ofereceria suporte às características e problemas inerentes a soluções orientadas a serviço. Para tal fim, lançou-se mão da taxonomia proposta por CARVALHO & AZEVEDO [17].

Segundo esta taxonomia, há três métodos que suportam a fase de construção de um ciclo de desenvolvimento de *software*, a saber: eXtreme Programming (XP), Pragmatic Programming (PP) e Feature-Driven Development (FDD). Analisando-se as práticas oferecidas por cada método, verifica-se que:

- Segundo LEE *et al.*, na proposta do FWSI (Framework for Web Services Implementation) [18], o XP é fortemente aplicável para melhorar a comunicação dos programadores entre si, e destes com os clientes, endereçando os pontos **(i)** e **(iii)**.
- O Planning Game do XP, segundo BECK [16], tem por propósito definir a ordem em que as funcionalidades serão implementadas, segundo as prioridades de negócio do cliente, dentro do domínio de seu problema, endereçando os problemas **(iii)**, **(ix)**, **(x)** e **(xii)**.
- O XP propõe a comunicação mais direta entre todos os envolvidos, endereçando os pontos **(i)**, **(iv)**, **(vi)**, **(ix)** e **(xi)**. No entanto, projetos SOA são de natureza distribuída, com equipes distribuídas geograficamente, o que impossibilita comunicação

face a face todo o tempo. Para contornar isso, KARSTEN & CANIZZO [15] e KIRCHER *et al.* [19] propõem o uso de ferramentas de videoconferência e compartilhamento de informações, além de eventos presenciais periódicos para compartilhar o andamento das equipes, informações e boas práticas;

- O XP endereça o ponto **(vii)** por meio de sua natureza iterativa e incremental. Os relatos de KARSTEN & CANIZZO [15] e MARANZATO, NEUBERT & HERCULANO [20] tratam de equipes distribuídas trabalhando de forma iterativa e incremental.

- O XP endereça o ponto **(xiii)** por meio da prática "Simple Design" [16], cujo propósito é garantir que o design da solução seja a solução mais simples que atende aos critérios de aceitação do produto, sendo estes definidos pelo cliente.

Em paralelo, verifica-se que o PP não oferece nenhuma prática que enderece os pontos citados, ao passo que o FDD endereça somente o ponto **(ix)**, através de sua prática de modelagem de domínio.

Assim, tomando-se por base os trabalhos publicados ao longo da pesquisa, verificou-se que: **(i)** As práticas do XP endereçam diversos dos problemas enumerados; **(ii)** Dentre os métodos ágeis, o XP é aquele que emprega o maior número de práticas compartilhadas entre estes métodos para a construção de serviços como identificado por CARVALHO & AZEVEDO [17].

Desta forma, a pesquisa se desenvolveu na direção de um método que fornecesse uma forma estruturada de endereçar os problemas inerentes a soluções orientadas a serviço por meio do uso de práticas ágeis do XP (eXtreme Programming) [16].

A necessidade de um método foi identificada a partir do trabalho de ABRANTES & TRAVASSOS [21], que afirmam ser importante que organizações que desejam adotar práticas ágeis sejam capazes de entender o que elas realmente precisam a partir de sua situação atual. Similarmente, LINDVALL *et al.* [22] afirmam que a utilização de alguma metodologia específica ou a adoção de determinadas práticas de desenvolvimento de software são consideradas não triviais e dependem de muitos fatores. Além disso, segundo NUNES [23], a falta de processos que se adequem às necessidades da organização, falta de integração entre processos de desenvolvimento e deficiência em processos de software contribuem decisivamente para o insucesso de projetos de software. Na mesma linha, NASSIR & SAHIBUDDIN [24] relatam que, segundo um estudo baseado em revisão sistemática, o uso de metodologias ou processos de desenvolvimento adequados é um fator crítico de sucesso para projetos de software.

1.3 Objetivo da pesquisa

Nosso trabalho, então, se direcionou a verificar se já havia na literatura algum método que endereçasse, de forma estruturada, o uso de métodos ágeis em soluções orientadas a serviço. Por "estruturado", entenda-se um método que defina fases, papéis, entregáveis e responsabilidades, podendo ser seguido por um time que deseje adotá-lo em uma organização, com suporte mínimo. Para tal fim, foi conduzido um estudo de mapeamento sistemático. O estudo foi conduzido utilizando motores de busca disponíveis no portal da CAPES, além de anais de conferências nas áreas de SOA e métodos ágeis, entre 2005 e 2012. As buscas em todas estas fontes foram feitas usando a mesma expressão de busca: *(SOA OU service-oriented OU SOSE OU SOC) E (XP OU "extreme programming")* - variando de acordo a sintaxe específica de cada ferramenta, conforme detalhado no Apêndice 6.

Como resultado, identificamos que diversos autores propõem o uso de práticas ágeis em diferentes aspectos do ciclo de vida SOA, mas nenhum endereçava numa mesma proposta questões inerentes ao uso combinado de métodos ágeis e SOA. Por exemplo, cada proposta endereçava um ou mais dos seguintes pontos de forma esparsa, nunca combinada: questões relacionadas a equipes distribuídas; mudanças em contratos de serviços; uso de critérios de aceitação e *User Stories*; entre outros aspectos ágeis deixados em aberto. Em resumo, há carência de métodos com detalhes suficientes para serem seguidos na prática, e que sejam também aderentes aos princípios ágeis e de orientação a serviços, conforme será apresentado mais adiante nesta dissertação.

Uma vez que não foi encontrado um método que endereçasse as questões necessárias, a pesquisa foi direcionada à proposição de um método, o qual foi apresentado por CARVALHO, AZEVEDO e SANTOS [25]. No entanto, após *feedbacks* obtidos e discussões subsequentes, entendemos que existem diversas práticas de XP e princípios de SOA, podendo estes ser combinados, de forma mais restrita ou relaxada de acordo com as necessidades de uma organização. Desta forma, percebemos que investir em um único método cujo objetivo fosse resolver qualquer situação, independente do contexto, seria infrutífero.

Assim, baseado na necessidade de um processo que leve em consideração o contexto de organizações e processos, chegamos ao objetivo final da pesquisa, de constituir uma Linha de Processo de Software (LPS). ARMBRUST *et al.* [26] definem LPS como um conjunto de processos com um conjunto de características gerenciadas que satisfazem necessidades específicas de uma organização particular e que são desenvolvidas a partir de um conjunto

básico de processos comuns de modo prescritivo. De acordo com NUNES, ROCHA & SANTOS [27], uma linha de processo de software (LPS) desempenha papel importante na constituição de um processo de software que a organização possa seguir. A flexibilidade inerente ao fato de a LPS considerar diversas variações do processo em cenários distintos possibilita com que se escolha um processo que seja mais adequado às necessidades da organização que a instancia, passando, assim, a levar em consideração o contexto dessa organização e seus processos pré-existentes. A LPS deste trabalho aponta os pontos de variação das necessidades identificadas, considerando a utilização de práticas ágeis na entrega de soluções orientadas a serviço, que enderecem os problemas e sejam aderentes aos princípios de SOA.

1.4 Fases da pesquisa

Para alcançar o objetivo desta dissertação as seguintes fases da pesquisa foram endereçadas:

1. Definir foco da pesquisa

Conforme demonstrado na Seção 1.2, diversos autores relatam problemas inerentes a soluções orientadas a serviço [18] [15] [6] [4]. Verificou-se que o XP [16] oferece práticas que endereçam a maioria destes problemas. O foco da pesquisa se direcionou ao uso de XP na entrega de soluções orientadas a serviço. Além disso, ao analisar o XP em relação a outros métodos para construção de serviços (PP e FDD), considerando os requisitos necessários para um processo de desenvolvimento (i.e., problemas apontados na literatura que precisam ser resolvidos por um processo de desenvolvimento orientado a serviços), observou-se que o XP é o método que melhor atende aos requisitos.

2. Pesquisar por métodos ou processos existentes que já endereçassem o objetivo e o foco definido para a dissertação

Foi conduzido um estudo de mapeamento sistemático (Apêndice A) com o objetivo de verificar a pré-existência de métodos que endereçassem o uso de XP em soluções orientadas a serviço. Este estudo identificou lacunas na literatura entre os trabalhos analisados e os princípios de SOA e XP, e concluiu que não havia na literatura trabalhos que endereçassem a combinação de SOA e XP no nível de detalhe adequado, conforme apresentado em CARVALHO, AZEVEDO e SANTOS [25].

3. Definir escopo da proposta

O método proposto por CARVALHO, AZEVEDO e SANTOS [25] endereça a fase de construção de um ciclo de desenvolvimento de soluções orientadas a serviço. Este método, no entanto, não leva em consideração o contexto das organizações onde seria aplicado. Segundo NUNES [23], a inadequação de processos às necessidades da organização é um fator crítico para o insucesso de projetos de software. Desta forma, nossa pesquisa foi ajustada para que o método fosse evoluído para uma Linha de Processo de Software, com variações que atendessem a diferentes realidades e contextos.

4. Elaboração da proposta:

As lacunas identificadas na literatura, em conjunto com os princípios de SOA e as práticas de XP, serviram de requisitos para a LPS, derivando, a seguir, as características a serem atendidas. Uma vez definidas estas características, foram criados componentes reutilizáveis endereçando tanto estes requisitos quanto necessidades de organizações que instanciassem esta LPS.

5. Avaliação da proposta:

A Linha de Processo proposta foi revisada através de revisão por pares por especialistas em Linhas de Processo de Software, Métodos Ágeis e SOA, com vieses de pesquisa e indústria. A Linha de Processo foi, posteriormente, adequada conforme resultados das avaliações.

1.5 Estrutura do trabalho

Este trabalho está dividido da seguinte forma. O Capítulo 1 é a presente introdução. O Capítulo 2 apresenta a fundamentação teórica, os trabalhos relacionados e apresenta o protocolo do estudo de mapeamento sistemático, bem como seus resultados. O Capítulo 3 apresenta a proposta deste trabalho, bem como sua avaliação por pares por especialistas nos assuntos de SOA e/ou metodos ageis. Finalmente, o Capítulo 4 apresenta a conclusão e trabalhos futuros.

2. Trabalhos relacionados

Este capítulo apresenta os conceitos básicos associados aos três principais assuntos desta dissertação: Arquitetura Orientada a Serviços (SOA, na sigla, em inglês), eXtreme Programming (XP) e Linhas de Processo de *Software* (LPS). Complementarmente, são apresentados trabalhos disponíveis na literatura que embasam esta dissertação e/ou se relacionam de alguma forma a abordagem proposta.

2.1 Arquitetura Orientada a Serviço (SOA)

Segundo HOHPE & WOOLF [28], organizações conduzem negócios por meio de centenas ou milhares de aplicações, de origens diversas: construídas sob medida, compradas como produto fechado de um fornecedor, parte de um sistema legado, ou ainda, como uma combinação de todos os anteriores, rodando sobre camadas diferentes de diversas plataformas distintas e que, nem sempre, se comunicam. Quanto maior a organização, maior a probabilidade de se encontrar em seu ambiente diversas soluções localizadas (automatizadas ou manuais) que resolvem problemas comuns a diversas áreas, pacotes proprietários de diferentes fornecedores e plataformas com o mesmo propósito, entre outras.

De forma geral, os usuários finais destes sistemas, aplicativos ou, simplesmente, soluções, não se preocupam nem têm ciência das fronteiras organizacionais envolvidas numa interação sua com uma função de negócio, também denominada capacidade de negócio. Eles sabem que aquela capacidade está disponível para seu uso por meio daquela solução, e simplesmente a executam, independentemente da quantidade de sistemas e parceiros de negócios envolvidos na concretização de todo aquele processo. Conforme o exemplo citado por HOHPE & WOOLF [28], uma atualização de endereço ou a verificação de pagamento de um cliente podem disparar solicitações a sistemas distintos, responsáveis, respectivamente, por relacionamento com clientes ou cobrança. Cada um destes sistemas pode disparar outras requisições subsequentes a diversas outras soluções das quais de-

pendem. Para que estas capacidades de negócio estejam disponíveis para usuários finais, todas essas soluções precisam ser integradas.

Conforme destaca JOSUTTIS [3], ao invés de sistemas individuais, o mundo caminha para grandes sistemas distribuídos, lidando com processos e integrações cada vez mais complexas. A abordagem de centralização e harmonização falha nos aspectos de distribuição e escalabilidade, demandando novas formas de trabalho, que lidem melhor com heterogeneidade e descentralização.

Em paralelo, a forma como consumidores encaram as organizações também vem mudando [2]. O perfil de consumo vem migrando de um orientado ao consumo de bens para um orientado ao consumo de serviços. LANKHORST cita como exemplo algo que se observa no dia a dia: os consumidores têm cada vez menos interesse em adquirir um carro ou uma impressora, e cada vez procuram mais serviços de impressão ou serviços móveis, por exemplo. Isto, por sua vez, leva as organizações a, paulatinamente, explorar a Internet para entrega de seus serviços, buscando novos dispositivos, tecnologias e infra-estruturas, visando melhorar a experiência do usuário com seu produto.

Dessa forma, passa a ser natural pensar em funções organizacionais na forma de serviços, os quais, por sua vez, podem ser definidos como funções de negócio auto-contidas. Estas funções podem integrar outros processos e serem implementadas por qualquer tecnologia em qualquer plataforma [3].

Alinhado às questões de reuso de processos ou funções de negócio, integração, descentralização e escalabilidade, o conceito de Arquitetura Orientada a Serviço (SOA) surgiu em meados da década de 1990 [3], estando, desde sempre, intimamente ligado a grandes sistemas distribuídos. SOA atua como facilitador no crescimento contínuo e inevitável destes sistemas, ao mesmo tempo em que contribui para manter sua escalabilidade e flexibilidade. SOA está de acordo que a única forma de conservar a flexibilidade de grandes sistemas distribuídos é suportando heterogeneidade, descentralização e tolerância a falhas.

Analogamente, segundo ERL [4], o conceito de SOA estabelece um modelo arquitetural que tem por objetivo aumentar a eficiência, agilidade e produtividade de uma organização, sendo este modelo composto de tecnologias, produtos, API's, infra-estrutura, entre outras partes. De fato, segundo JOSUTTIS [3] reforça, SOA não depende de uma tecnologia ou produto específico, podendo ser implementado usando diversas tecnologias em conjunto, como, por exemplo, Web Services, CORBA, MQ, Tibco. O aumento na eficiência, agilidade e produtividade de uma organização é atingido pelo posicionamento de seus serviços como canal primário para realização de metas estratégicas.

Em resumo, conforme JOSUTTIS [3], ERL [4] e LANKHORST [2], SOA é um paradigma arquitetural para lidar com processos de negócio distribuídos por sistemas heterogêneos, sejam eles legados ou recém-construídos. SOA tem o objetivo de oferecer funções de negócio de forma escalável, obtendo maior eficiência, agilidade, reuso, flexibilidade e produtividade por meio de seu suporte a heterogeneidade, descentralização e tolerância a falhas.

De acordo com ERL [4], o conceito de orientação a serviço traz consigo alguns princípios de projeto. ERL [4] define oito princípios, explicados a seguir, os quais, quando aplicados ao *design* de uma unidade lógica, resultam em soluções orientadas a serviço.

2.1.1Princípios de Orientação a Serviço

2.1.1.1Contrato Padronizado de Serviço (*Standardized Service Contract*)

Segundo ERL [4], um serviço deve, por princípio, expressar por meio de um contrato seu propósito, capacidades, restrições e informações semânticas que o responsável pelo serviço deseje tornar públicas. Estas informações são descritas seguindo normas de *design* compartilhadas por todos os serviços contidos em um dado repositório.

Este contrato pode ser expresso por meio de um conjunto de documentos de descrição do serviço, ficando cada um responsável por descrever uma parte do serviço. Este conjunto é composto por documentos a serem consumidos em tempo de execução e/ou documentos textuais (i.e., "não-técnicos") que complementam seus detalhes técnicos.

Há diversas formas de se expressar um documento técnico de descrição de serviço. Uma forma bastante comum é por meio de uma API (Application Programming Interface), onde o consumidor de uma dada função deve ter acesso a uma representação local do contrato (também conhecido como *proxy*), para poder se comunicar com o serviço remoto. Esta forma de comunicação é implementada por diversos *frameworks* baseados em Remote Procedure Call (RPC), como, por exemplo, RMI, CORBA e DCOM. Por sua vez, na implementação de serviços por meio de *Web Services*, um serviço é descrito por meio de uma linguagem não-proprietária (Web Service Description Language - WSDL), baseada em XML que dispensa o uso de *proxies*.

Independente do formato, os objetivos do contrato são [4]:

- Aumentar o nível de interoperabilidade entre serviços, por meio de modelos de dados consistentes, o que reduz a necessidade de transformações de dados;

- Permitir que o propósito e as capacidades do serviço sejam facilmente inteligíveis, favorecendo sua interpretação e previsibilidade;

De forma geral, os objetivos acima reforçam que um contrato se destina a leitura e interpretação tanto por máquinas quanto por seres humanos. Mais do que simplesmente favorecer o entendimento humano do detalhamento textual de suas capacidades e restrições, estes objetivos visam facilitar a automação da descoberta e uso de um dado serviço, perseguindo os objetivos finais de maximização do reuso e minimização de desenvolvimento customizado para atender a um objetivo de negócio. Para atingir esses objetivos, ERL [4] propõe algumas práticas.

A primeira prática proposta diz respeito à governança da representação dos dados trafegados pelos serviços. Muitas vezes um serviço agregará operações que trabalham sobre um dado nicho de negócio, podendo, usar informações idênticas, ou com afinidade de negócio entre si, como entrada ou saída em suas operações. Considere o exemplo de uma companhia aérea fictícia, que expõe um serviço de dados de passageiros frequentes com duas operações: uma para recuperar os últimos vôos de um passageiro, e outra para recuperar os dados pessoais de um passageiro, conforme o exemplo em pseudo-código na Figura 2.1.

```
ServicoDadosPassageiroFrequente{
   ultimosVoos(UltimosVoosRequest): UltimosVoosResponse
    dadosPessoais(DadosPessoaisRequest):
        DadosPessoaisResponse
    passageirosComUltimoNome(
        PassageirosComUltimoNomeRequest):
            PassageirosComUltimoNomeResponse
}
```

Figura 2.1: Exemplo operação de serviço.

Considere ainda as seguintes representações das mensagens trafegadas pelo serviço apresentadas na 2.2.

Note-se que os objetos `UltimosVoosRequest` e `DadosPessoaisRequest` são bastante semelhantes entre si. Estes, por sua vez, guardam alguma semelhança com o objeto `DadosPessoaisResponse`. Considere-se um cenário onde uma dada função ou processo de negócio precise, por exemplo, saber os dados pessoais de todos os passageiros frequentes com sobrenome “Silva”. Para tal, será necessário invocar a operação `passageirosComUltimoNome`, e, para cada instância de `Passageiro` retornada, convertê-la para `DadosPessoaisRequest`, a fim de poder invocar a operação

```
UltimosVoosRequest{
    registroPassageiroFrequente: Inteiro
    cpf: String
}

UltimosVoosResponse{
    numerosVoos: Lista<String>
}

DadosPessoaisRequest{
    registroPassageiroFrequente: Inteiro
    cpf: String
}

DadosPessoaisResponse{
    registroPassageiroFrequente: Inteiro
    cpf: String
    primeiroNome: String
    ultimoNome: String
    endereco: String
}

PassageirosComUltimoNomeRequest{
    ultimoNome: String
}

PassageirosComUltimoNomeResponse{
    cpf's: Lista<String>
}
```

Figura 2.2: Estrutura das mensagens trafegadas pelo serviço.

`dadosPessoais` para cada passageiro e, ao final, extrair o `cpf` de cada `Passageiro` para enfim retornar a informação.

Ainda que num primeiro momento essas semelhanças não despertem muita atenção, elas passam a representar um desafio quando são considera as dificuldades para interoperabilidade e automação na orquestração destas chamadas. Diversos objetos contêm informações semelhantes, requerendo intervenção humana para possibilitar as diversas transformações que atenderão à requisição.

A proposta de ERL [4] no que se refere a governança da representação dos dados para atender ao princípio de contrato padronizado diz respeito a estabelecer normas e padrões que possibilitem maior reuso de objetos ou mensagens entre as operações, de forma a

minimizar o número de transformações necessárias e favorecer a comunicação automática entre as operações.

Na proposta de ERL [4], uma possível estruturação das operações e mensagens poderia ser feita como apresentado na Figura 2.3. Esta representação evita replicação de dados.

```
ServicoDadosPassageiroFrequente{
    ultimosVoos(Passageiro): ColecaoString
    dadosPessoais(Passageiro): Passageiro
    passageirosComUltimoNome(Passageiro): ColecaoString
}

ColecaoString{
    colecao: Lista<String>
}

Passageiro {
    registroPassageiroFrequente: Inteiro
    cpf: String
    primeiroNome: String
    ultimoNome: String
    endereco: String
}
```

Figura 2.3: Estrutura mais enxuta de mensagem.

Esta forma de trabalho reduz a complexidade de administração do modelo de objetos, além de favorecer a interoperabilidade e automação de chamadas. Obviamente este é apenas um exemplo feito para demonstrar um caso extremo que ilustra o conceito, mas empresas reais enfrentam este tipo de problema cotidianamente.

A segunda prática proposta diz respeito a governança das políticas de serviço.

O padrão WS-Policy[1] define algumas políticas padrão, implementadas por todos os processadores de políticas. No entanto, políticas podem ser customizadas e incorporadas ao contrato, eventualmente sendo validadas somente pela lógica do serviço quando de sua invocação. Assim, de forma análoga à primeira prática, ERL [4] propõe que políticas customizadas sejam modularizadas e estruturadas de forma que todos os serviços de um dado repositório implementem as mesmas políticas, facilitando, mais uma vez, a interoperabilidade e automação do uso de serviços de um mesmo repositório.

[1] `http://www.w3.org/TR/2006/WD-ws-policy-primer-20061221/`

2.1.1.2Baixo Acoplamento (*Service Loose Coupling*)

De acordo com ERL [4], um serviço é fracamente acoplado se as dependências entre seu contrato, sua implementação e seus consumidores são mínimas. O objetivo deste princípio é possibilitar um ambiente onde os serviços e seus consumidores podem ser evoluídos com impacto mínimo entre si. O acoplamento entre estes elementos é inerente ao desenvolvimento de serviços, a meta é minimizar seu grau. Há dois tipos de acoplamento [4]: um que se refere ao contrato e outro que se refere ao consumidor. São cinco os tipos de acoplamento relativos a contrato [4]:

- **Lógica para Contrato**: Um serviço cuja lógica tem alto acoplamento com seu contrato oferecerá maior confiabilidade, melhor desempenho e maior facilidade de customização do serviço, além de garantir o cumprimento dos padrões de projeto estabelecidos na organização. Ao contrário dos demais tipos, este é considerado desejável, devendo ser buscado seu maior grau.

- **Contrato para Lógica**: De forma inversa ao primeiro tipo de acoplamento, não é desejável que um contrato tenha sua lógica implementada anteriormente ao seu contrato. Uma vez que o serviço ou seu contrato estejam publicados, haverá consumidores dependendo do contrato naquela forma. Assim, modificações em um contrato trazem consigo impacto direto em consumidores diversos como, por exemplo: outros serviços internos, parceiros de negócio. Desta forma, um serviço cujo contrato é derivado de sua implementação estará sujeito a modificações com maior frequência, uma vez que a lógica de negócio de um serviço está, fundamentalmente, sujeita a modificações a qualquer momento, de forma a refletir de forma ágil as mudanças no negócio da organização. Este tipo de acoplamento, quando em grau alto, tende a inibir a evolução contínua do serviço.

- **Contrato para Tecnologia**: Um alto grau de acoplamento entre um contrato e a tecnologia que o suporta fere o objetivo da evolução contínua com impacto mínimo entre serviços ou para consumidores, na medida em que, se o contrato especifica o uso de algum componente ou protocolo proprietário, uma mudança de fornecedor resultará em impacto para os consumidores daquele serviço.

- **Contrato para Implementação**: Este tipo de acoplamento de concretiza quando o contrato referencia diretamente detalhes de sua implementação ou seu ambiente de execução, como, por exemplo, caminhos de arquivos, caminhos de rede, nomes ou versões de API's internas, nomes de tabelas. De forma análoga ao acoplamento do tipo ***Contrato para Tecnologia***, este tipo de acoplamento aumenta as dificuldades

de evolução de um serviço, na medida em que mudanças de fornecedor de uma API, nomes de tabelas ou servidores, por exemplo, acarretam impacto aos consumidores do serviço.

- **Contrato para Funcionalidade Externa**: Este tipo de acoplamento muitas vezes ocorre quando um serviço é projetado para encapsular um processo de negócio ou um parceiro de negócio em específico. Com isso, modificações no serviço fornecido pelo parceiro ou no processo de negócio acarretam modificações no contrato e, por conseguinte, impacto sobre os seus consumidores.

Os tipos de acoplamento que se referem ao consumidor se dividem em duas categorias [4]:

- **Consumidor para Implementação**: Um contrato não é mandatório para uso de um serviço por um consumidor. O consumidor pode descartar o uso do contrato e se conectar diretamente à implementação, por motivos diversos como, por exemplo, economia de recursos ou facilidade de *design*. Quando isto ocorre, este tipo de acoplamento se materializa, trazendo impactos diretos ao consumidor quando a lógica do serviço é modificada.
- **Consumidor para Contrato**: Este é um tipo de acoplamento recomendado e desejável, uma vez que oferece o maior grau de independência entre um serviço e seu consumidor.

Analogamente, KRAFZIG, BANKE & SLAMA [29] propõem outros tipos de acoplamento, sumarizados na Tabela 2.1:

2.1.1.3Abstração (*Service Abstraction*)

Conforme definido por ERL [4], o contrato de um serviço deve limitar as informações que torna públicas, reduzindo ao que seja estritamente essencial. De forma mais ampla, o contrato só deve expor o essencial, e as informações públicas de um serviço devem se limitar ao que está exposto no contrato. Esta abordagem visa maximizar o acoplamento do tipo ***Consumidor para Contrato***, anteriormente descrito como recomendado e desejável. Desta forma, tornando públicas somente as informações essenciais sobre o serviço e omitindo informações sobre detalhes de funcionamento interno (tecnologia, lógica, caminhos de rede, servidores), o provedor do serviço tende a não ter consumidores fortemente acoplados à implementação, o que favorece a evolução do serviço com impacto mínimo a seus consumidores.

Tabela 2.1: Tipos de acoplamento segundo KRAFZIG, BANKE & SLAMA [29]

Tipo	**Alto Acoplamento**	**Baixo Acoplamento**
Conexões físicas	Ponto a ponto	Via mediador
Estilo de comunicação	Síncrono	Assíncrono
Modelo de dados	Tipos complexos padronizados	Somente tipos simples
Tipagem	Forte	Fraca
Padrão de interação	Navegação por árvores de objetos complexos	Mensagens auto-contidas
Controle do processamento lógico	Central	Distribuído
Ligação (*Binding*)	Estático	Dinâmico
Plataforma	Fortes dependências da plataforma	Independente de plataforma
Transações	*Two phase commit*	*Compensation*
Deployment	Simultâneo	Em tempos distintos
Versionamento	Evoluções explícitas	Evoluções ímplicitas

ERL [4] define quatro tipos de abstração:

- **Tecnológica**: diz respeito a omissão de detalhes tecnológicos de implementação do serviço, como, por exemplo, linguagem de programação ou bibliotecas utilizadas na construção do serviço. As informações expostas devem, idealmente, se limitar às essenciais ao uso do serviço, como, por exemplo, protocolo aceito pelo serviço (ex.: SOAP [30], no caso de *Web Services*, ou MQ [31], no caso de serviços assíncronos).

- **Funcional**: Visa limitar as informações disponíveis sobre as capacidades ou funções executadas por um serviço, de acordo com os critérios da organização provedora deste serviço. Por exemplo, voltando ao caso da empresa aérea fictícia, há um serviço que disponibiliza operações para acessoa dados pessoais de passageiros frequentes. Em paralelo, a equipe de desenvolvimento do serviço optou por guardar na base de dados somente o CEP de cada passageiro, reutilizando um serviço dos Correios para informar o endereço completo de um passageiro frequente. Este mesmo serviço dos Correios é reutilizado em diversas outras funcionalidades, e poderia ser exposto como operação pela empresa aérea. A **Abstração Funcional** é

usada quando o contrato expõe a operação de busca de dados pessoais e omite a capacidade de buscar endereços a partir de um CEP, pois o negócio da empresa diz respeito a informações sobre seus passageiros e vôos, e não a informações de logradouros.

- **Lógica**: Esta forma de abstração visa omitir do consumidor detalhes lógicos de implementação do serviço, como, por exemplo, algoritmos ou mecanismos de *logging*. Por meio desta abstração, o contrato refere-se a **o quê** o serviço faz, abstraindo **como** é feito.

- **Qualidade de Serviço (QoS)**: limita a exposição de informações sobre informações não-funcionais do serviço, como capacidades e limitações de processamento. Um exemplo de informação não-funcional pode ser um aviso de que caso a mensagem de retorno exceda um dado tamanho, será enviada por e-mail ao usuário ao invés de ser retornada sincronamente. Ou um aviso de que o serviço fica fora do ar durante finais de semana para economia de recursos.

O nível de abstração de contratos pode ser medido em três dimensões, segundo ERL [4]:

Níveis de abstração de conteúdo do contrato: Dividida em **detalhado**, **conciso**, **otimizado** e **misto**, esta dimensão avalia a exposição de detalhes funcionais no contrato. Busca-se o nível **otimizado**, onde o contrato não expõe informações internas como restrições de entrada ou regras de negócio.

Níveis de controle de acesso: Esta dimensão avalia a exposição de detalhes tecnológicos e lógicos por meio de três níveis: **aberto**, **controlado** e **sem acesso**. O acesso a artefatos é classificado como código-fonte, especificações de projeto e outros documentos que expõe o funcionamento interno do serviço.

2.1.1.4Reuso (*Service Reusability*)

Um serviço é um ativo de uma organização [4]. Logo, serviço deve gerar retorno sobre o investimento feito em sua aquisição. Quanto maior o seu uso pela organização nas mais variadas situações, seja de forma direta (por meio de invocações sucessivas) ou indireta (a partir de seu uso como parte da resolução de outros problemas), maior o retorno obtido sobre seu investimento. Desta forma, este princípio define que serviços devem conter e expressar lógica agnóstica de contexto, de modo a serem posicionados como recursos corporativos reutilizáveis.

Os objetivos de negócio por trás deste princípio englobam dimensões diversas: (i) econômica: por envolver retorno sobre investimento; (ii) agilidade de negócio: pela criação mais rápida de novas soluções a partir de lógicas pré-existentes; e, (iii) de organização de trabalho: ao buscar a implantação de um modelo de serviços agnósticos, onde os serviços contidos no repositório organizacional atingem alto nível de reuso.

Segundo ERL [4], há quatro características de *design* suportadas por este princípio, as quais podem ser usadas isoladamente ou combinadas, conforme explicado a seguir. Note-se que, do ponto de vista de reuso, serviços podem ser classificados de duas formas: (i) "*agnóstico*": quando fornece lógica que não é específica ou atrelada a nenhum processo de negócio da corporação; e, (ii) "*reutilizável*" quando pode ser usado na automação de diversos processos de negócio.

- **Serviço agnóstico**: alcançada quando a lógica do serviço é suficientemente agnóstica em relação a contexto de qualquer cenário de uso;
- **Serviço genérico**: diz respeito a serviços cuja lógica é suficientemente genérica, podendo facilitar seu uso em cenários diversos, servindo a consumidores com propósitos diversos;
- **Contrato extensível**: se refere ao contrato ser flexível o suficiente para permitir um gama abrangente de mensagens de entrada e saída;
- **Concorrência**: define que serviços devem ser projetados para suportar acesso simultâneo por múltiplos consumidores;

Ainda segundo ERL [4], há três formas de se medir o reuso planejado para um serviço, usadas como guia durante o *design* do mesmo:

- **Reuso tático**: se aplica quando o as características do projeto de desenvolvimento especificam uma janela de tempo estreita, ou quando os serviços demandados não têm perspectiva de reuso. Neste caso, os serviços são projetados para atender somente a necessidade imediata.
- **Reuso direcionado**: utilizada quando há garantia de potencial de reuso de uma ou mais partes do serviço sendo desenvolvido. Neste cenário, o projeto dos serviços deve considerar as necessidades imediatas e futuras conhecidas.
- **Reuso completo**: seu uso só é recomendado quando a organização possui um repositório de serviços bem definido, aliado a um planejamento claro de suas necessidades e as oportunidades de reuso entre elas. Nesta abordagem, os serviços são

projetados para oferecer um conjunto abrangente de funcionalidades de negócio, mesmo que não haja consumidores imediatos.

2.1.1.5Autonomia (*Service Autonomy*)

O nível de autonomia de um serviço, conforme descrito por ERL [4], trata do relacionamento entre um serviço e seu ambiente de execução. Quanto maior o nível de controle do primeiro sobre o último, maior a autonomia do serviço. Este princípio procura atender a dois objetivos de negócio: sua confiabilidade e a previsibilidade de seu comportamento. Quanto menos agentes externos influenciarem um serviço, maior sua confiabilidade e previsibilidade.

O princípio de autonomia diz respeito não só a dependências entre o serviços e os ambientes físico e lógico de sua plataforma de execução, mas também à sobreposição funcional entre suas capacidades e as dos demais integrantes do repositório da organização. Há dois tipos de autonomia [4]:

- **Autonomia de tempo de execução**: Este tipo diz respeito ao nível de controle que o serviço tem sobre sua lógica e as respostas que produz. Um serviço que produz respostas sem depender de outros serviços ou outras fontes de dados terá grande autonomia de tempo de execução. Analogamente, um serviço que colabora com outros em uma composição e depende de capacidades externas às suas fronteiras para produzir uma resposta, terá menor autonomia. Neste caso, o nível de autonomia do serviço será dado em função dos níveis de autonomia dos demais serviços.
- **Autonomia de tempo de *design***: Este tipo de autonomia varia com o nível de governança e controle permitido ao responsável por um serviço. Por exemplo, a partir do momento em que um serviço é publicado, diversos consumidores passam a depender daquele contrato no formato em que está. A partir deste momento, o nível de autonomia de tempo de *design* decai, pois o responsável pelo serviço não pode mais evolui-lo e modificá-lo de forma arbitrária, pois, por exemplo, as mudanças terão de levar em consideração impacto aos consumidores, compatibilidade retroativa. Analogamente, quanto maior a autonomia de tempo de *design* de um serviço, maior será sua autonomia de tempo de execução, uma vez que haverá menos dependências para com consumidores e serviços que participam de alguma composição.

O nível de autonomia de um serviço pode ter relação direta com suas capacidades não-funcionais. Tome-se como exemplo um cenário onde um serviço **S1** é demandado para

desenvolvimento, tendo como critério de aceitação não-funcional suportar um tempo de resposta inferior a 5 segundos com 500 usuários concorrentes. Além disso, as informações de potencial de reuso ditam que **S1** deve reutilizar um serviço **S2**, onde **S2** tem como requisitos suportar no máximo 200 usuários concorrentes, e seu tempo máximo de resposta é de 10 segundos. Em tal cenário, a autonomia de **S1** é dada em função de **S2**, o que implica dizer que **S1** não pode garantir o tempo de resposta solicitado para a carga especificada. Nesta situação, será necessário escolher entre alterar o critério de aceitação não-funcional de **S1**, o que pode não ser viável, dados os compromissos assumidos com parceiros de negócio, ou solicitar a evolução de **S2** para acomodar a nova carga extra.

ERL [4] sugere que o nível de autonomia de um serviço conste do seu contrato, de forma a explicitar a potenciais consumidores suas capacidades não-funcionais e não comprometer a autonomia destes, ainda que, em algum nível, vá contra o princípio de **Abstração**. ERL [4] considera que o benefício trazido por esta informação extra no contrato é maior do que o prejuízo causado pela violação da **Abstração** do serviço.

ERL [4] define quatro medidas de autonomia de serviço a serem usadas como informação pública no contrato:

- **Contrato**: A autonomia de contrato visa delimitar as capacidades declaradas nos contratos de serviços. Esta medida varia com a sobreposição de responsabilidades declaradas entre serviços de um mesmo repositório: se diversos serviços de um mesmo repositório declaram se destinar a obter informações pessoais de passageiros frequentes, por exemplo, então há baixo nível de autonomia entre estes contratos. Note-se que esta medida só engloba os limites funcionais expostos no contrato. Se diversos serviços expressam responsabilidades diferentes mas compartilham lógicas internas, sua medida de autonomia de contrato não será influenciada, uma vez que não há sobreposição funcional.

- **Compartilhada**: Serviços são categorizados nesta medida quando compartilham lógica e recursos com outros serviços ou outros ativos da organização. Esta é uma categoria bastante comum, uma vez que é comum em organizações o uso de serviços para encapsular legados ou outras aplicações. Assim, segundo ERL [4], esta categoria é particularmente útil para indicar que consumidores do serviço podem enfrentar concorrência de outros consumidores, com objetivos diversos.

- **Lógica de serviço**: Esta categoria indica que a lógica do serviço é independente, mas há compartilhamento de recursos de armazenamento de dados com outras partes da organização. De acordo com ERL [4], esta forma autonomia é bastante comum

no desenvolvimento de novos serviços, com problemas conhecidos como: *lock* de registros de banco de dados, tempo elevado de execução de *queries*, baixa escalabilidade, tempos de resposta inconsistentes, entre outros, devido a concorrência por recursos entre o serviço e outros componentes da organização.

- **Pura**: neste caso, tanto a lógica do serviço quanto seu canal de acesso a dados são dedicados, não há compartilhamento de recursos, o que torna o serviço mais autônomo e, portanto, mais previsível, customizável e escalável. Em tempo de *design*, a autonomia deste serviço permite evolução mais livre, com menos preocupações relativas a integrações com serviços parceiros.

2.1.1.6Ausência de Estado (*Service Statelessness*)

Como ativo corporativo que é, um serviço deve oferecer máximo retorno sobre o seu investimento. O retorno sobre o investimento é diretamente proporcional a regularidade com que o serviço é invocado e reutilizado em novas composições. Assim, de forma a não comprometer sua escalabilidade e sua disponibilidade, este serviço deve minimizar o seu consumo de recursos, visando, por fim, atender ao máximo de requisições concorrentes com a mesma qualidade de serviço. Essa minimização no consumo de recursos é alcançada pela delegação do gerenciamento de informações de estado, quando este gerenciamento for necessário [4].

Em outras palavras, este princípio trata de maximização da escalabilidade de um serviço. Quanto mais escalável um serviço, mais favorável será seu reuso. A máxima escalabilidade é alcançada pela redução de consumo de recursos gastos com gerenciamento de estado. O gerenciamento de estado pode ser inevitável a um serviço, o que não impede que o gasto de recursos com essa tarefa seja reduzido por meio de sua delegação a um mecanismo especialista nesta funcionalidade.

Há quatro tipos de informações sobre estado que podem ser usadas para descrever um serviço e suas características, segundo ERL [4]:

- **Informações sobre estado primário**: Descrevem os estados possíveis de um serviço, no que se refere ao atendimento de requisições: **ativo** ou **inativo**;
- **Informações sobre condições de estado primário**: Definem como o serviço gerencia informações de estado quando **ativo**: ***stateless*** ou ***stateful***. Enquanto no primeiro o serviço não retém informações de estado, no segundo o serviço faz gerenciamento ativo de seu estado.

As informações de estado se dividem em três categorias:

- **Sessão**: são informações utilizadas tipicamente para manter ou reestabelecer a conexão entre um serviço e seu consumidor;
- **Contexto**: são informações trocadas entre serviços que cooperam para executar uma funcionalidade de negócio;
- **Negócio**: ao contrário dos anteriores, informações desse tipo não expressam estado nem são usadas para fins de conectividade, mas representam informações de negócio que podem precisar sem temporariamente armazenadas para posterior reuso;

ERL [4] também define diversas categorias destinadas a classificação de serviços quanto ao seu nível de gerenciamento de informações de estado, informações estas que, qual o princípio de **Autonomia**, visam explicitar capacidades não-funcionais a consumidores, em detrimento de sua **Abstração**. São elas:

- **Gerenciamento não-delegado**: usada quando apresenta baixíssima ausência de estado, indica que: (i) o serviço se mantém ativo mesmo quando desnecessário e (ii) contém uma quantidade significativa de lógica para retenção de informações de estado de consumidores, como, por exemplo, informações de sessão e contexto;
- **Delegação parcial em memória**: aplica-se a serviços que se mantém ativos o tempo todo, lidam com informações de estado, mas as delegam e a algum mecanismo externo de gerenciamento de estado quando não são necessárias, recuperando-as de volta conforme a necessidade;
- **Delegação arquitetural parcial**: ao contrário das anteriores, serviços nesta classificação não se mantém ativos o tempo todo, transitando para o estado **inativo** quando necessário. Porém, de forma análoga ao **gerenciamento não-delegado**, não delega informações de estado;
- **Delegação arquitetural completa**: serviços desse tipo lançam mão da delegação de informações de estado sempre que possível, enquanto ativos;
- **Delegação interna**: nesta classificação, não há gerenciamento de estado pelo serviço, toda informação de estado é delegada para um mecanismo segregado de gerenciamento de estado, interno ao seu funcionamento, abstrato ao consumidor.

2.1.1.7Descoberta de serviço (*Service Discoverability*)

Um serviço é um ativo corporativo cujo reuso aumenta o retorno sobre seu investimento. Para ser reusado e, consequentemente, aumentar o retorno sobre o investimento feito sobre si, é preciso saber que o serviço existe e a que se destina. Este princípio, segundo ERL [4], tem por objetivo assegurar que o propósito e capacidades de um serviço estejam expostos adequadamente, favorecendo sua descoberta e interpretação, tanto por humanos quanto por máquinas, os quais se tornam, então, responsáveis por decidir se o serviço encontrado se adequa às suas necessidades.

Para tanto, informações sobre propósito e capacidades devem ser registradas como meta-informações no contrato do serviço, permitindo sua pesquisa e descoberta por meio de algum mecanismo, para posterior interpretação.

ERL [4] descreve dois tipos de meta-dados que podem ser usados para incrementar o contrato e descrever o melhor o serviço no que tange as funções suportadas e os requisitos não-funcionais suportados, aumentando, assim, sua possibilidade de descoberta num repositório por humanos ou máquinas.

- **Meta dados funcionais**: dizem respeito a padronização das informações funcionais tratadas pelo princípio de **Contrato padronizado**. A combinação de ambos os princípios visa garantir que os serviços de um repositório documentam suas informações funcionais e capacidades de forma clara e padronizada, favorecendo sua descoberta a partir de uma *query* executada por humanos ou máquinas.

- **Meta dados de qualidade de serviço**: essas informações complementam os meta dados funcionais, dando ao consumidor a possibilidade de analisar uma gama de serviços funcionalmente próximos e escolher aquele que atende melhor a suas limitações não-funcionais. Voltando ao exemplo fictício do repositório de serviços da companhia aérea, considere-se um cenário onde haja dois serviços que expõe dados de passageiros: um dedicado a passageiros frequentes, cujos meta-dados indicam um tempo de resposta constante de 500ms; outro cujo propósito é fornecer dados de qualquer passageiro que já tenha voado com a companhia aérea, com tempo de resposta variável entre 200 milissegundos e 10 segundos, dependendo do quão específicos ou abrangentes são os parâmetros de entrada. Numa situação onde é necessário recuperar os dados de um passageiro frequente a partir de um parâmetro direcionado (por exemplo, sua *primary key* no banco de dados), e onde haja uma restrição de tempo de resposta inferior a 300 milissegundos, o uso do serviço de dados de todos os passageiros passa a ser mais viável, apesar de haver um serviço

dedicado a dados de passageiros frequentes.

A medida do quanto um serviço é aderente a este princípio é de difícil obtenção. ERL [4] sugere o uso de *checklists* para aferir níveis fundamentais de descoberta, mas admite que a avaliação da qualidade dos meta-dados de descoberta é subjetiva, dependendo de amplo entendimento do propósito e capacidades do serviço, em conjunto com conhecimentos sobre seu ambiente de execução. Ele sugere ainda o uso de um vocabulário padronizado, lançando mão de palavras-chaves que facilitem a pesquisa e aumentem o potencial de descoberta de um serviço.

2.2 Métodos Ágeis e XP

Segundo STRODE [32], há diversos métodos de desenvolvimento de sistemas: estruturado, orientado a objetos, RAD, entre outros. Métodos ágeis são um grupo de métodos publicados entre o final da década de 1990 e início da década de 2000, por autores diversos com interesses em comum. Estes autores compartilhavam o embasamento em experiências práticas e a necessidade de desenvolvimento iterativo e incremental focado na entrega rápida de *software* com qualidade. Estes interesses e embasamentos compartilhados deram origem a interseções entre os métodos destes autores. COCKBURN [33], por exemplo, cita diversas características em comum com o XP, como a busca por produtividade crescente e valorização da disciplina. Estes métodos foram inicialmente publicados isoladamente e agrupados sob a denominação de "métodos ágeis" em 2001, com a publicação do manifesto ágil [9].

STRODE [32] faz uma consolidação de 12 métodos ágeis publicados entre 1997 e 2003 e apresenta uma série de características em comum entre os mesmos:

- Endereçam problemas de negócio;
- Baseiam-se na prática;
- Têm foco no desenvolvimento iterativo e incremental;
- Endereçam projetos sob constante mudança;
- Preconizam envolvimento ativo do usuário;
- Enfatizam trabalho em equipe;
- Enfatizam a necessidade de *feedback* e aprendizado;

- Equipes têm poder de decisão;
- Equipes devem ser pequenas, entre 3 a 10 programadores;
- Documentação deve ser mínima;
- *Software* funcionando é o principal produto do desenvolvimento;

A seguir são apresentados estes métodos e uma breve descrição de suas aplicações.

2.2.1Dynamic Systems Development Method (DSDM)

Dynamic Systems Development method (DSDM) é apresentado como o primeiro método verdadeiramente ágil de desenvolvimento de *software* [34]. O DSDM surgiu em 1994 como um *framework* para desenvolvimento rápido de aplicações (RAD), mantido pela DSDM Consortium [35]. Sua aplicabilidade se estende ao gerenciamento de todo o desenvolvimento, tendo um ciclo de vida que engloba estudo de viabilidade, estudo do negócio e fases iterativas e incrementais de prototipação, construção e implantação. Os estudos de viabilidade e do negócio só ocorrem uma vez, antes do início da construção, sendo esta dividida em iterações de tempo fixo, a cargo de equipes pequenas.

A idéia fundamental do método, segundo ABRAHAMSSON *et al.* [35] [34], é que, ao invés de fixar o escopo do produto e ajustar tempo e recursos para entregá-lo, se fixe o tempo e os recursos, e o escopo a ser entregue seja planejado de acordo com estas capacidades.

O DSDM descreve 9 princípios:

1. Envolvimento ativo do usuário;
2. Equipes devem ter autorização para tomar decisões;
3. Foco na entrega contínua de produtos;
4. A aderência ao propósito de negócio é critério essencial de aceitação dos entregáveis;
5. Desenvolvimento iterativo e incremental;
6. Mudanças ao longo do desenvolvimento devem ser reversíveis;
7. Os requisitos são congelados somente no alto nível;

8. Testes devem estar integrados ao ciclo de vida de desenvolvimento;

9. Todos os interessados devem colaborar e cooperar;

Além disso, os requisitos são priorizados segundo a técnica das regras de Moscou, também conhecidas como "MoSCoW Rules" [36], na sigla em inglês:

- ***Must have* (imprescindíveis)**: os requisitos desta classificação devem ser entregues, sob pena do sistema não funcionar corretamente sem estes;
- ***Should have* (importantes)**: estes são requisitos importantes e agregam valor significativo ao negócio, mas podem ser omitidos se por restrição de tempo entrarem no caminho de requisitos *must have*;
- ***Could have* (desejáveis)**: melhoram o sistema sem impedir seu funcionamento e serão construídos se o tempo e os recursos permitirem;
- ***Want to have* (supérfluos)**: usualmente são requisitos que atendem a um grupo limitado de usuários e agregam pouco valor, sendo entregues conforme disponibilidade de tempo e recursos;

2.2.2Crystal Methods

Crystal é uma família de práticas de gerenciamento de projetos, proposta por COCKBURN [37], cuja premissa básica é que o tamanho e criticidade do projeto ditem o peso do gerenciamento e coordenação a serem aplicados [35]. STRODE [32] o define como um conjunto de métodos para projetar uma metodologia aplicável a um projeto em específico.

O método Crystal prevê um conjunto de práticas a serem seguidas por toda a família, ficando a cargo de cada metodologia escolher a melhor maneira de implementá-las, de acordo com as necessidades de rigor inerentes aos cenários que buscam cobrir. Há diversos níveis de Crystal, variando com a complexidade. Para projetos de 2 a 6 pessoas, usa-se o Crystal Clear. O Crystal Yellow se destina a projetos de 6 a 20 pessoas. Crystal Orange endereça 20 a 40 pessoas. Há outras cores determinando a complexidade do projeto: Crystal Red, Magenta, Blue, *etc*. Os produtos previstos são os seguintes:

- ***Policy standards* (políticas-padrão)**: práticas a serem aplicadas durante o desenvolvimento:
  - Entrega incremental com frequência regular;

  - Acompanhamento de progresso por meio de marcos definidos por entregas ou decisões importantes;
  - Envolvimento direto do usuário;
  - Teste de regressão automatizado;
  - Duas revisões de usuário por *Release*;
  - Oficinas de ajuste da metodologia e do produto no início e meio de cada iteração;

- ***Work products*** **(produtos de trabalho)**: evidências de trabalho a serem geradas:
  - Planejamento de *Releases*;
  - Modelo de objetos comuns;
  - Manual do usuário;
  - Casos de teste;
  - Código de migração;
  - Requisitos;
  - Documentação de interface;

- ***Local matters*** **(necessidades específicas)**: templates que devem ser preenchidos de acordo com as necessidades de cada projeto:
  - Padrão de codificação;
  - Padrão de teste de regressão;
  - Padrão de interface;

- ***Tools*** **(ferramentas)**: não há necessidade de nenhuma ferramenta específica, mas o método Crystal Clear, por exemplo, requer um compilador, um gerenciador de configuração e quadros brancos com funcionalidade de impressão, para armazenamento de material gerado em reuniões. Já o Crystal Orange requer ferramentas para teste, acompanhamento de projeto e medição de performance.

- ***Standards*** **(padrões)**: definem padrões a serem usados pelo projeto, como, por exemplo, padrões de notação, projeto, formatação e qualidade.

- ***Activities*** **(atividades)**: diversas atividades podem ser executadas durante o andamento de uma iteração, de acordo com o método da família Crystal a ser utilizado. De maneira geral, as seguintes atividades são executadas: planejamento da iteração, construção, demonstração, revisão, monitoramento, ajuste da metodologia.

2.2.3Rational Unified Process (RUP)

ABRAHAMSSON, SALO & RONKAINEN [35] define o RUP como uma abordagem iterativa para o desenvolvimento de sistemas orientados a objetos, fortemente acoplada a modelagem por UML e requisitos definidos por casos de uso.

O RUP, resumidamente, é dividido em quatro fases:

- ***Inception* (iniciação)**: o escopo, os critérios de aceitação e os casos de uso críticos são identificados, a arquitetura candidata é desenhada e o custo e duração de todo o projeto são estimados;
- ***Elaboration* (elaboração)**: descreve os casos de uso, a arquitetura de *software* e é criado um protótipo executável da arquitetura;
- ***Construction* (construção)**: o *software* é construído e testado;
- ***Transition* (transição)**: esta fase consiste em testes de aceitação, treinamento e implantação do produto;

Durante estas fases, nove fluxos de trabalho estão em execução: modelagem de negócio, requisitos, análise e projeto, implementação, teste, gerência de configuração e mudança, gerência do projeto e manutenção do ambiente do RUP.

O RUP é baseado em seis práticas:

1. *Software* desenvolvido de maneira iterativa;
2. Os requisitos são gerenciados;
3. Arquitetura baseada em componentes;
4. Modelagem visual do *software*;
5. Qualidade verificável do *software*;
6. As mudanças devem ser controladas;

Segundo KRUCHTEN [38] e ABRAHAMSSON, SALO & RONKAINEN [35], o RUP pode ser adotado "*out of the box*" ou customizado conforme a necessidade.

2.2.4Extreme Programming (XP)

Ao contrário dos métodos apresentados até aqui, o eXtreme Programming (XP) foca no estágio de construção do ciclo de desenvolvimento de *software*. Proposto por Kent Beck, Ward Cunningham e Ron Jeffries, XP é um conjunto de boas práticas de engenharia de *software* cujo objetivo é estabelecer um desenvolvimento bem sucedido, apesar de requisitos vagos ou em constante mudança [34]. XP baseia-se nos valores de simplicidade, comunicação, *feedback*, coragem e respeito [39]. A idéia é juntar toda a equipe de desenvolvimento, utilizar práticas simples, com *feedback* suficiente para possibilitar à equipe saber onde estão e ajustar as práticas à situação de trabalho. STRODE [32] define o XP como sendo uma metodologia para desenvolvimento de *software* em ambientes de alta incidência de mudanças usando equipes pequenas e técnicas padrão de engenharia de *software* para satisfazer as necessidades do cliente e manter equipes efetivas.

O XP tem algumas práticas principais:

- ***Whole Team***: não há especialistas, todos os envolvidos contribuem da melhor forma possível;
- ***Planning Game***: o cliente participa do planejamento das *Releases* e das iterações, definindo as prioridades de negócio e acompanhando a complexidade das histórias;
- ***Small Releases***: as entregas são feitas em pequenas partes;
- ***Customer Tests***: o cliente define testes de aceitação a serem automatizados, e nestes estão incluídos o conjunto de testes de regressão;
- ***Simple Design***: o projeto do *software* é incremental, atendendo às necessidades acumuladas até o momento;
- ***Pair Programming***: todo o código é produzido em pares, garantindo que o código é revisado por pelo menos um programador;
- ***Test-Driven Development***: todo código de produção escrito deve ter um teste que ateste seu funcionamento, garantindo *feedback* rápido quando um código novo afeta o resto do sistema;
- ***Design Improvement***: o código deve ser constantemente refatorado, removendo duplicidades, aumentando a coesão e diminuindo o acoplamento;
- ***Continuous Integration***: o sistema deve estar totalmente integrado o tempo todo, com várias construções correndo ao longo do dia;

- ***Collective Code Ownership***: não há "dono" de um pedaço do sistema; todos trabalham em todas as partes do sistema e podem atuar nelas quando necessário;
- ***Coding Standards***: o código segue uma padronização para que pareça ter sido escrito por uma única e competente pessoa, favorecendo a propriedade coletiva;
- ***Metaphor***: o time compartilha a mesma visão do que o sistema deve fazer;
- ***Sustainable Pace***: os times devem trabalhar num ritmo duro e que seja sustentável no longo prazo;

  Segundo BECK [16], o XP deve ser aplicado preferencialmente a times de 3 a 20 pessoas, em um grande ambiente físico formado por pequenos cubículos, preferencialmente com todos os envolvidos no mesmo andar.

2.2.5Adaptive Software Development (ASD)

ASM é um *framework* proposto por Jim Highsmith que foca nos problemas inerentes ao desenvolvimento de sistemas grandes e complexos [35]. STRODE [32] o define como um *framework* para gerenciamento de projetos de desenvolvimento de *software* sob intensa pressão de prazo e com requisitos mudando constantemente. Seu ciclo de vida propõe três fases [40]:

- ***Speculation***: fixa prazos e objetivos da iteração, e define o planejamento baseado nas funcionalidades ou componentes;
- ***Collaboration***: desenvolvimento concorrente dos componentes;
- ***Learn***: apresenta as funcionalidades desenvolvidas aos interessados;

Ao início do projeto, é estabelecida a missão e todo o desenvolvimento visa cumpri-la. Além disso, o processo é iterativo e incremental, e prega a presença do cliente em sessões de JAD (*Joint Application Development*). ASM tem seis características principais:

1. Orientado a missão;
2. Orientado a componentes;
3. Iterativo;
4. Iterações de tempo fixo;
5. Tolerante a mudanças;
6. Guiado pelos riscos;

2.2.6Scrum

Segundo ABRAHAMSSON *et al.*[35] [34], o Scrum é um processo empírico de gerência de desenvolvimento de *software* em ambientes voláteis, baseado em flexibilidade, adap-tabilidade e produtividade. O Scrum não define nenhuma técnica específica de construção de *software*, o objetivo é identificar e remover deficiências e impedimentos no processo de desenvolvimento. STRODE [32] define o Scrum como sendo basicamente uma metodologia de gerenciamento de projetos para desenvolvimento iterativo.

A sua idéia principal, segundo ABRAHAMSSON, SALO & RONKAINEN [35], é que o desenvolvimento de sistemas envolve diversas variáveis de cunhos ambiental e técnico, sujeitas a mudanças durante o processo, tornando-o imprevisível e complexo, requerendo flexibilidade para tornar o processo mais responsivo a mudanças. Como resultado, é produzido um sistema que é útil quando entregue.

O Scrum propõe três fases:

1. ***Pre-game***: subdividida em planning e architecture, esta fase define a lista de requisitos a serem implementados (*Product Backlog*) e a arquitetura em alto nível que atende a estes requisitos;
2. ***Game***: dividida em *sprints* ou iterações, esta fase inclui as fases tradicionais do desenvolvimento de *software*: requisitos, análise, projeto, construção e entrega. A arquitetura e o projeto do sistema evoluem com o andamento das *sprints*;
3. ***Post-game***: quando se chega a um acordo de que os requisitos necessários à *Release* foram finalizados, esta fase é iniciada, onde são feitos os testes de integração e documentação.

O Scrum requer algumas práticas de gerenciamento do desenvolvimento:

1. ***Product backlog***: é a lista de requisitos a serem atendidos para que o projeto seja visto como concluído. Esta lista é atualizada e repriorizada a cada iteração;
2. ***Effort estimation***: processo iterativo, onde as estimativas de esforço são atualizadas conforme novas informações sobre os requisitos são disponibilizadas;
3. ***Sprint***: é o período de trabalho durante a fase Game, gerando um novo incremento executável do produto ao seu final;

4. ***Sprint planning meeting***: reunião de duas fases que define a parte do backlog a ser implementada durante a próxima *sprint*, e discute como trabalhar para alcançá-la;

5. ***Sprint backlog***: ponto de partida da *sprint*. Ele é o conjunto de tarefas do *product backlog* que será trabalhado durante a *sprint* seguinte, permanecendo estável pela duração da *sprint*;

6. ***Daily Scrum meeting***: reunião curta, idealmente com 15 minutos de duração, que visa discutir o que foi feito no dia anterior, o que será feito no dia, e os impedimentos e deficiências a serem removidos;

7. ***Sprint review meeting***: ao final da sprint, os resultados são apresentados e avaliados, e é decidido qual será o caminho seguido na próxima sprint;

Acompanhando a tendência de outros métodos ágeis, o Scrum recomenda equipes de 3 a 10 integrantes. Frequentemente, o XP é usado em conjunto com Scrum para formar um pacote integrado para equipes de desenvolvimento de *software*.

2.2.7Pragmatic Programming (PP)

Segundo ABRAHAMSSON *et al.* [35] [34], Pragmatic Programming (PP) não é exatamente uma metodologia, e sim um conjunto de boas práticas de programação que resolvem problemas do dia a dia e são focadas em desenvolvimento iterativo e incremental, testes rigorosos e projeto centrado no usuário.

Apesar de ser constituído de 70 práticas, a filosofia por trás destas pode ser resumida em 6 pontos:

1. Assuma a responsabilidade por suas ações;

2. Não aceite código ou projeto ruim;

3. Assuma um papel ativo na introdução de mudanças onde achar necessário;

4. Faça *software* que satisfaça seu cliente, mas saiba quando parar;

5. Aumente seu conhecimento constantemente;

6. Melhore suas habilidades de comunicação.

2.2.8Internet Speed Development (ISD)

Segundo ABRAHAMSSON *et al.* [34], ISD não é uma metodologia, mas uma abordagem de desenvolvimento ágil de *software* aplicável a situações onde o *software* precisa ser entregue rapidamente, requerendo ciclos curtos de desenvolvimento. ISD propõe um *framework* para atacar o problema de gerenciamento de *Releases* rápidas. Segundo BASKERVILLE & RAMESH [41], três fatores causais tiveram grande importância em determinar a necessidade desta abordagem:

- Corrida desenfreada para atender ao mercado;
- Ambiente de mercado novo e único;
- Falta de experiência no desenvolvimento de *software* sob as condições impostas por este ambiente;

BASKERVILLE & RAMESH [41] apresentam oito práticas que caracterizam o ISD:

- ***Develop in parallel***: funcionalidades são desenvolvidas em paralelo de modo a aumentar a velocidade de entrega de produtos;
- ***Release more often***: ao entregar com velocidade, a priorização das funcionalidades pode mudar com mais facilidade, e permite também que uma nova funcionalidade só seja agregada quando o mercado a demandar;
- ***Depend on tools***: são usadas ferramentas que fornecem soluções de produtividade pré-prontas, evitando a necessidade de customização de ferramentas de trabalho;
- ***Implant customers in the development environment***: projetos ISD dependem do envolvimento do cliente, para a priorização correta do que deve ser encaixado nas *Releases* de acordo com a necessidade de negócio e na velocidade adequada;
- ***Establish a stable architecture***: ISD prega o estabelecimento de uma boa arquitetura fixa, provendo similaridade entre os *Releases*, maximizando o reuso e evitando a necessidade de refazer projetos devido a código ruim;
- ***Assemble and reuse components***: o reuso de componentes é imprescindível em ISD. Com frequência, a velocidade adequada só é alcançada se o reuso é praticado, evitando construir *software* novo;

- ***Ignore maintenance***: o curto tempo de vida de projetos ISD leva as equipes a desenvolver uma mentalidade onde não há preocupação com o custo de manutenção do código;
- ***Tailor the methodology daily***: com a necessidade de manter uma velocidade alta, equipes ISD tendem a customizar suas práticas e usar somente o básico necessário de processos. Tem-se a tendência de ignorar fases ou tarefas que impeçam entregar *software* dentro do prazo, mesmo sob o risco de produzir código de qualidade mais baixa;

BASKERVILLE & RAMESH [41] apresentam também 7 princípios de ISD, alinhados as suas práticas e compatíveis com os princípios ágeis:

1. ***Accept multiple valid approaches***: ainda que cada componente do sistema tenha sido desenvolvido sob uma metodologia diferente, o comportamento do sistema não pode ser comprometido;
2. ***Engage the customer***: ao fazer *Releases* frequentes com a colaboração do cliente, este tem satisfação imediata ao ver suas idéias concretizadas em um novo *Release*;
3. ***Accommodate requirements change***: todas as práticas do ISD, de uma maneira ou de outra, buscam atender a este princípio. Mudanças de última hora são sempre bem vindas e todas as práticas visam acomodar as mudanças de modo a entregar um produto que atende as necessidades do cliente;
4. ***Build on successful experience***: guarde informações sobre experiências bem sucedidas e as reutilize todo o tempo;
5. ***Develop good teamwork***: a escolha certa de pessoas em colaboração com um cliente próximo usando a metodologia adequada em *Releases* curtas permitem maior entendimento do que precisa ser feito e como trabalhar em conjunto;
6. ***Conform to project environment constraints***: uma arquitetura estável e escalável, em combinação com *Releases* frequentes, permitem aos desenvolvedores se adaptar melhor a mudanças no ambiente que envolve o projeto;
7. ***Prepare for unexpected consequences from software process innovation***: o uso de componentes prontos, arquitetura estável e ferramentas estáveis promovem a produtividade, mas, em algum momento, podem passar a ser um problema. As equipes devem estar aptas a mitigar isto por meio da inovação, estabelecendo rapidamente uma nova maneira de trabalhar;

Por fim, BASKERVILLE & RAMESH [41] apresentam algumas diferenças fundamentais em relação a outros métodos de desenvolvimento de *software*:

1. A velocidade de desenvolvimento é o principal direcionador do processo de desenvolvimento. Custo e qualidade são secundários, onde o custo do *software* é mais um item a compor os custos operacionais, e a qualidade é algo negociável e medido pela funcionalidade e disponibilidade do produto;
2. Não há início ou fim de projetos, projetos são operações contínuas;
3. A manutenção não é tratada separadamente, mas embutida nos ciclos de *Release*;
4. O gerenciamento de recursos humanos é diferente no sentido de que o conhecimento é mais tácito, tornando pessoas menos intercambiáveis e requerendo pessoas com maior iniciativa, criatividade e coragem.

2.2.9Agile Modeling (AM)

Assim como o XP e o PP, AM não é uma metodologia que engloba todo o ciclo de vida de desenvolvimento de *software*. Segundo ABRAHAMSSON *et al.*[35] [34], AM correponde a um conjunto de práticas que visa implantar uma mudança cultural onde a modelagem de um sistema seja feita segundo os mesmos princípios ágeis do XP: presença do cliente, equipes pequenas, forte comunicação e colaboração, equipes multidisciplinares, documentação mínima e necessária.

AM propõe 11 práticas centrais e 8 suplementares, que podem ser divididas nas categorias:

- Modelagem iterativa e incremental;
- Trabalho em equipe;
- Simplicidade;
- Validação;
- Produtividade;
- Documentação;
- Motivação;

AM observa que outros métodos ágeis cobrem a parte de modelagem e propõem a integração com outras metodologias de desenvolvimento, como XP e RUP.

2.2.10Feature Driven Development (FDD)

De acordo com ABRAHAMSSON *et al.* [35] [34], FDD é uma metodologia de desenvolvimento de *software* orientada a processos para a construção de sistemas críticos ao negócio. O FDD não ataca todas as etapas do ciclo de vida de desenvolvimento de *software*, sendo focado nas fases de projeto e construção com ênfase em qualidade, apesar de ter sido projetado para se integrar com outros processos que cubram todo o ciclo de vida.

O processo é dividido em 5 fases, sendo as 2 últimas direcionadas ao trabalho iterativo e incremental:

1. ***Develop an overall model***: a partir da documentação dos requisitos funcionais, é gerado um modelo do domínio do negócio;
2. ***Build a features list***: usando a documentação de requisitos e o modelo de negócio, é gerada uma lista com as funcionalidades enxergadas pelo cliente no sistema;
3. ***Plan by feature***: a lista de funções é priorizada e são atribuídas responsabilidades entre os programadores para cada área de negócio do sistema;
4. ***Design by feature, Build by feature***: essas 2 fases são direcionadas ao projeto e construção de cada funcionalidade de forma iterativa e incremental, podendo ocorrer também com times em paralelo.

FDD envolve as seguintes práticas:

- Modelagem dos objetos de domínio;
- Desenvolvimento por funcionalidade;
- Propriedade individual de cada classe de domínio;
- Equipes divididas por funcionalidade;
- Inspeção de artefatos;
- *Builds* constantes;
- Gerência de configuração;
- Reporte de progresso;

2.2.11 Open Source Software Development (OSSD)

Como apontado por ABRAHAMSSON, SALO & RONKAINEN [35], OSSD não é uma compilação de práticas de desenvolvimento de *software*, mas pode ser melhor descrito em termos de diferentes licenças para distribuição de *software*, e como uma maneira colaborativa de indivíduos geograficamente dispersos produzirem *software* com incrementos pequenos e contínuos. Alguns pesquisadores sustentam que um projeto OSSD representa uma organização virtual.

O processo seguido por um projeto OSSD não tem os mesmos mecanismos de coordenação, projeto e acompanhamento de outros métodos ágeis. De fato, sua semelhança com esses métodos se dá muito mais no campo da filosofia e práticas. Tipicamente um projeto OSSD consiste das seguintes fases:

- Descoberta do problema
- Procura de voluntários
- Identificação da solução
- Desenvolvimento e teste
- Revisão de código
- Publicação e documentação do código
- Gerência de *Release*

O gerenciamento do processo também é bastante diferente, conforme delineado por ABRAHAMSSON, SALO & RONKAINEN [35]:

- Os sistemas são construídos por voluntários;
- O trabalho é auto assinalado;
- Não há projeto explícito do sistema;
- Não há planos de projeto ou lista de entregáveis;
- O sistema cresce por meio de pequenos incrementos;
- O código é testado com frequência;

2.2.12Lean

O termo *lean*, numa tradução livre, significa "enxuto", e vem da expressão *lean manufacturing* (manufatura enxuta). Segundo POPPENDIECK & POPPENDIECK [42], Lean é a aplicação dos princípios do Sistema de Desenvolvimento de Produtos da Toyota ao desenvolvimento de sistema, os quais, quando corretamente aplicados, resultarão em *software* de qualidade que é desenvolvido com velocidade e a baixo custo. Conforme POPPENDIECK & POPPENDIECK [43], a filosofia do Lean é composta por 22 "ferramentas de pensamento" que ajudam os líderes de equipes de desenvolvimento a adaptar práticas ágeis as suas realidades. POPPENDIECK & POPPENDIECK [42] listam sete princípios da "manufatura enxuta" aplicáveis ao desenvolvimento de *software*:

- ***Eliminate waste***: descubra o que o cliente quer, construa e entregue exatamente o que ele quer, virtualmente imediatamente. Qualquer coisa que impeça a satisfação imediata da necessidade do cliente, qualquer coisa que não agregue valor percebido pelo cliente ao produto é desperdício;
- ***Amplify learning***: desenvolvimento de *software* é um processo de aprendizado, com equipes grandes e resultados complexos. É natural testar variações até chegar a um ritmo de produtividade previsível que produza um produto estável em produção. A melhor abordagem para melhorar um ambiente de desenvolvimento de *software* é amplificar as possibilidades de aprendizado;
- ***Decide as late as possible***: se a incerteza for alta, atrase as decisões pelo máximo de tempo possível, de modo que quando esta for tomada, as especulações já tenham se tornado fatos, o futuro já esteja mais próximo e previsível;
- ***Deliver as fast as possible***: entregas rápidas permitem ao cliente ter o que precisa mais rápido e atrasar decisões sobre o que é necessário até que ele tenha mais conhecimento. Sem velocidade, não é possível atrasar decisões, o que acarreta não ter *feedback* confiável. Quanto mais curto o ciclo de incrementos, mais se aprende e se conhece o produto;
- ***Empower the team***: envolva nas decisões técnicas todos os envolvidos em fazer o trabalho, tanto desenvolvedores quanto especialistas do negócio. Desse modo se combina o conhecimento dos detalhes ao poder de múltiplas mentes analisando o problema a ser resolvido;
- ***Build integrity in***: o *software* deve ser construído de forma a manter sua integridade ao longo do tempo, dos pontos de vista conceitual, da utilidade, usabilidade,

manutenibilidade, adaptabilidade e extensibilidade.

- ***See the whole***: mantenha o foco na otimização do produto como um todo, não deixe que os especialistas se preocupem só com suas partes;

2.2.13Sumário

Analisando as características dos métodos citados, propomos a classificação dos mesmos nas categorias apresentadas na Tabela 2.2.

Tabela 2.2: Classificação dos métodos ágeis catalogados por STRODE [32] , segundo a taxonomia proposta por CARVALHO & AZEVEDO [17].

Gerenciamento de ciclo de vida	Scrum, RUP, DSDM, ASD, ISD
***Frameworks* de práticas**	Lean, Crystal
Construção de *software*	XP, FDD, PP
Desenvolvimento distribuído de *software*	OSSD
Modelagem de *software*	AM

Esta dissertação tem por objetivo tratar a construção de soluções orientadas a serviço por meio de práticas ágeis. A classificação apresentada na Tabela 2.2 identifica os métodos ágeis mais voltados à construção de *software*, o que torna XP, FDD e PP elegíveis ao uso em nossa abordagem. No entanto, conforme pesquisa conduzida por MELO *et al.* [44], XP é considerado mais popular do que os outros dois métodos voltados a construção. Complementarmente, conforme levantamento publicado por CARVALHO & AZEVEDO [17], XP é o método que tem maior número de interseções com os demais métodos, cobrindo um maior espectro do ciclo de vida como um todo. De fato, conforme CARVALHO & AZEVEDO [17], o XP usa aproximadamente 45% das práticas ágeis catalogadas na literatura, seguida por Feature-Driven Development e Pragmatic Programming, com 17% e 12%, respectivamente. Por exemplo, assim como o Scrum, XP recomenda estimativas de esforço e planejamento das iterações. Da mesma forma, assim como DSDM e OSSD, o XP preconiza o uso de testes de forma integrada ao ciclo de vida.

Assim, considerando os aspectos apresentados acima, XP foi escolhido como método ágil a servir de base em nossa proposta de desenvolvimento ágil de serviços.

2.3 Linha de Processo de *Software* (LPS)

Para entender o conceito de Linha de Processo de *Software*, é necessário revisitar o conceito de Linha de Produto de *Software*, descrito por PARNAS [45], na qual aquele se baseia. Segundo PARNAS [45] e ARMBRUST *et al.* [26], uma família de *software* é um conjunto de *softwares* que compartilha uma extensa gama de características, de forma que a estratégia de construção do *software* deve considerar um projeto mais cuidadoso, visando maior reuso entre os componentes internos. Este maior reuso, em última instância, leva à conversão de uma coleção de produtos de *software* em uma linha de produtos de *software*, onde estes compartilham um núcleo que contém um conjunto de funcionalidades que satisfazem necessidades específicas, seja de uma indústria, mercado ou missão. Esta forma de organização leva à redução de esforços de desenvolvimento e manutenção do *software*, o que acaba por tornar lucrativo o maior custo de planejamento inicial.

Analogamente, o conceito de Linha de Processo de *Software* (LPS) visa a redução e centralização dos processos (ou objetos, no vocabulário de Linha de Processo de *Software*) a serem evoluídos por uma organização, buscando maior alinhamento ao seu negócio ou missão. O propósito de uma LPS é reutilizar componentes de processo, os quais, em última instância, representam conhecimento especializado sobre como executar um determinado processo de negócio. Assim, o conceito de LPS não diz respeito somente à centralização e ao reuso, mas também à manutenção do conhecimento especializado gerado em uma organização.

ARMBRUST *et al.* [26] definem Linha de Processo de *Software* como um conjunto de processos com um conjunto de características gerenciadas que satisfazem necessidades específicas de uma organização particular e que são desenvolvidas a partir de um conjunto básico de processos comuns de modo prescritivo. Segundo TEIXEIRA [46], LPS é uma abordagem para reuso de processos de *software*, baseada em padrões, arquiteturas, repositórios, *templates* e gerência de configuração, na qual novos processos, mais específicos, são definidos a partir de processos mais abrangentes, pré-existentes. Segundo BARRETO, MURTA & ROCHA [47], uma Linha de Processo de *Software* é uma forma de se instanciar um processo, a partir de um conjunto pré-existente de componentes de processo, onde cada instância apresenta características próprias. Desta forma, uma Linha de Processo de *Software* equivale conceitualmente a uma linha de produtos, onde o produto final é *software*. A Figura 2.4 apresenta um exemplo de uma Linha de Processo de *Software*. Os principais conceitos de LPS são apresentados na Seção 2.3.1 e estão resumidos na Tabela 2.3.

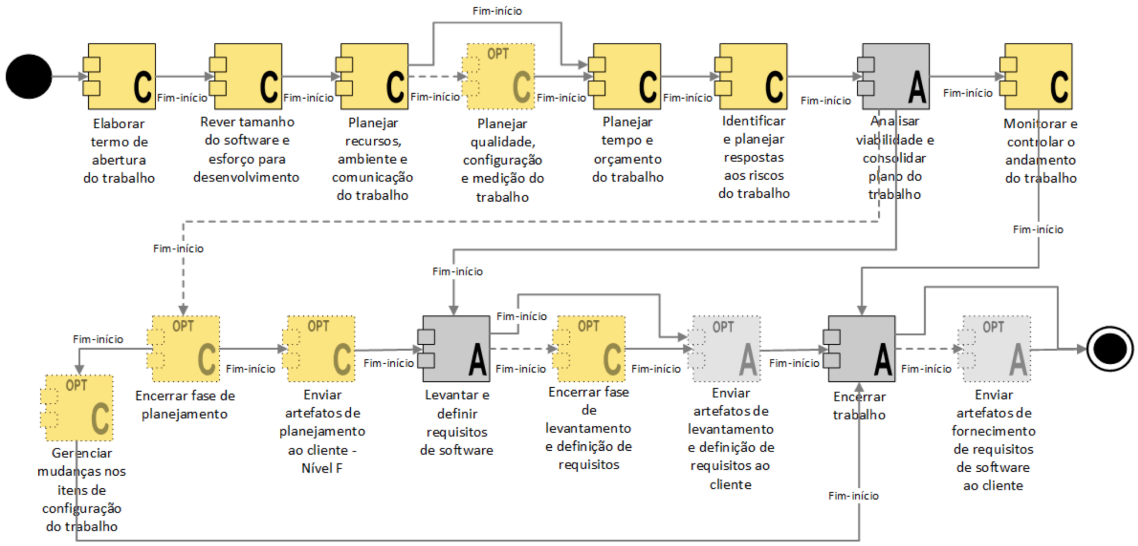


Figura 2.4: Exemplo de Linha de Processo de *Software* [27]

De forma geral, segundo ARMBRUST *et al.* [26], uma LPS tem dois objetivos:

1. Prever o número de variações possíveis dos processos de uma organização, de forma que todas as suas atividades sejam suportadas e o esforço de evolução seja mínimo;
2. Antecipar necessidades futuras, de forma que, quando uma dada necessidade se concretize, o processo que a atende esteja disponível.

BARRETO [48] vai além dos pontos citados por ARMBRUST *et al.* [26], e explicita outros objetivos que guiam a decisão de uma organização pela definição de Linhas de Processo de *Software*. Segundo BARRETO [48] *apud* ESTUBLIER & DAMI [49], processos são representações do conhecimento adquirido e acumulado por uma organização, sendo, portanto, ativos importantes para esta. Em paralelo, conforme BARRETO [48] *apud* COSTA *et al.* [50] e XU [51], a reutilização de processos se destaca por promover a melhoria contínua destes, aproveitando informações históricas, conhecimentos e experiências adquiridas em projetos anteriores para evoluir e disseminar suas representações. Assim, a reutilização de processos permite a uma organização obter vantagens competitivas em diversas frentes, tais como:

- Ganhos de ordem econômica, pela redução do esforço necessário para o desenvolvimento de novos processos e novos *softwares*;
- Aumento na qualidade do *software* produzido;
- Aumento na qualidade da representação do conhecimento da organização por meio de um processo.

Segundo CARDOSO [52], uma revisão sistemática de literatura conduzida por BARRETO [48] demonstra que há diversas técnicas de reutilização de processos relatadas na literatura, onde "Componentes de Processo", "Linhas de Processo" e "Padrões de processo" tem maior incidência, nesta ordem, correspondendo a 59% dos resultados. No entanto, segundo estudo experimental conduzido por BARRETO [48], a definição do processo para um projeto por meio de derivação de uma LPS apresentou economia de 25% do tempo, em comparação ao uso de componentes de processo. De forma complementar, o ferramental de suporte ao uso de LPS está em constante amadurecimento [52]. Além disso, segundo NUNES, ROCHA & SANTOS [27], a flexibilidade concedida por uma LPS, na medida em que considera diversas variações do processo em cenários distintos, possibilita com que se escolha um processo que seja mais adequado às necessidades de uma organização, considerando seus requisitos em um dado contexto.

2.3.1Conceitos principais

Segundo CARDOSO [52], na definição de processos baseada em reutilização, alguns conceitos principais se destacam, descritos nas seções a seguir.

2.3.1.1Elemento de processo

O primeiro conceito fundamental relacionado à definição de processos baseada em reutilização é o de **elemento de processo**. Este conceito é definido como sendo a agregação e encapsulamento de informações e comportamentos de um processo em um dado nível de granularidade. Um elemento de processo representa uma ação desempenhada por um processo, e pode ser classificado como uma **atividade** ou um **componente de processo**.

Uma atividade representa uma ação que pode ser utilizada para se atingir um resultado. Uma atividade pode também ser definida como um agrupamento de ações relacionadas de mais baixo nível.

Um componente de processo, assim como a atividade, visa atingir um resultado. No entanto, ao contrário daquela, um componente pode ser definido para reutilização, sendo então definido por CARDOSO [52] *apud* BARRETO [48] como a unidade básica utilizada na definição de processos baseados em reutilização. Um componente de processo tipicamente respeita uma ou mais características do conjunto a seguir:

1. Deve ser relevante no que se refere ao negócio que o processo representa;
2. Deve ser a representação de um subprocesso relevante;

3. Sua medição deve ser relevante;
4. Deve ter sua estabilidade e desempenho analisados;
5. Pode encapsular dentro de si um conjunto de outros elementos de processo;
6. Deve admitir variantes;

Um componente de processo pode ser classificado como **concreto** (não admite variabilidade, uma vez que já foi totalmente definido) ou **abstrato** (admite variabilidade, pois sua definição não está completa, e pode ser realizado de diferentes maneiras, através de diferentes componentes concretos). Ambos os tipos de componentes admitem a definição de uma **arquitetura interna**, ainda que o componente abstrato não seja executável, uma vez que é indefinido por natureza. A arquitetura interna de um componente concreto será composta pela execução de outros componentes concretos ou atividades. Um componente abstrato, por sua vez, pode não possuir uma arquitetura interna, ou ser decomposto em outros componentes abstratos ou componentes opcionais. Em outras palavras, um componente abstrato só possuirá uma arquitetura interna se esta não o definir por completo.

2.3.1.2Arquitetura de processo

Segundo CARDOSO [52] *apud* BARRETO [48], uma arquitetura de processo é uma estrutura que define a ordenação, as interfaces e as dependências entre os elementos de processo, podendo ser utilizada para variar o nível de granularidade em que um processo é definido. A ordenação entre dois elementos de processo é representada por meio de uma **conexão de elementos de processo**. Há cinco tipos de conexão, segundo CARDOSO [52] *apud* BARRETO [48], podendo estas serem também opcionais:

- **simples**: pode ser usada para ligar o início da arquitetura ao primeiro elemento de processo, ou o último elemento ao fim da arquitetura.
- **fim-início**: indica que a execução de um elemento terminou, podendo a execução do elemento seguinte ser iniciada.
- **início-início**: representa o início concorrente de dois ou mais elementos de processo.
- **fim-fim**: indica que os elementos conectados devem terminar juntos.
- **início-fim**: indica que, quando o primeiro elemento se inicia, o elemento ao qual está conectado deve terminar.

2.3.1.3Linha de processo

Uma **linha de processo** é uma arquitetura de processo independente, significando que ela não integra a estrutura interna de um componente de processo. A variabilidade da linha de processo se dá pelo uso de componentes abstratos, cujas variantes serão executadas de acordo com as características de processo selecionadas.

2.3.1.4Característica de processo

Segundo CARDOSO [52] *apud* BARRETO [48], uma característica é um aspecto ou característica com a qual o processo deve ser compatibilizado. Tanto um componente quanto uma linha de processo podem ser derivados de uma característica do processo, o que, por sua vez, pode influenciar a execução de determinadas variantes dos mesmos.

As características de um processo podem apresentar relações de dependência ou conflito, indicando que, quando uma dada característica *C1* for selecionada, a seleção da característica *C2* se dará pelo tipo de dependência: se foram conflitantes, somente a *C1* será selecionada; caso contrário, *C2* será também selecionada.

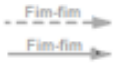
2.3.1.5Representação gráfica

A Tabela 2.3 apresenta um sumário dos conceitos apresentados, em forma de notação gráfica.

Tabela 2.3: Notação gráfica proposta por BARRETO [48]

Notação	**Nome do elemento**	**Descrição**
	Componente de processo concreto obrigatório	Representa elementos não configuráveis.
	Componente de processo concreto opcional	Representa opcionalidade de elementos não configuráveis.
	Componente de processo abstrato obrigatório	Representa pontos de variação.

	Componente de processo abstrato opcional	Representa opcionalidade de pontos de variação.
	Atividade obrigatória	Representa atividade obrigatória em um arquitetura de processos.
	Atividade opcional	Representa atividade opcional em um arquitetura de processos.
	Tipo de conexão opcional entre elementos de processo	Representa relacionamento opcional entre elementos de processos.
	Tipo de conexão obrigatória entre elementos de processo	Representa relacionamento obrigatório entre elementos de processos.
	Tipo de conexão Fim-início entre elementos de processo	Indica que ao término da execução do elemento de processo origem, a execução do elemento destino pode ser iniciada.
	Tipo de conexão Início-início entre elementos de processo	Indica que quando se inicia a execução do elemento de processo origem, a execução do elemento de processo destino também pode se iniciar.

	Tipo de conexão Fim-fim entre elementos de processo	Indica que ao término da execução do elemento de processo origem, a execução do elemento de processo destino também deve se encerrar.
	Tipo de conexão Início-fim entre elementos de processo	Indica que quando se iniciar execução do elemento de processo origem, a execução do elemento de processo destino deve se encerrar.
	Item início da arquitetura	Representa o ponto de início dentro de uma arquitetura.
	Item fim da arquitetura	Representa o ponto de encerramento dentro de uma arquitetura.

2.3.2Definição de processos para SOA e XP

De acordo com JOSUTTIS [3], SOA é um paradigma que consiste, dentre outros elementos, de políticas e processos para lidar com a heterogeneidade de grandes sistemas distribuídos com diferentes proprietários. Em outras palavras, SOA é uma abordagem que preconiza fundamentalmente processos bem definidos. Como qualquer abordagem ou paradigma, SOA tem limitações, conforme apresentado na Seção 1.2. Essas limitações podem ser superadas pela sua combinação a outras abordagens.

Conforme apresentado na Seção 2.2.13, XP é uma abordagem para *design* e construção de *software*, sem que, no entanto, haja definição formal de processos a serem executados durante o desenvolvimento. Em paralelo, apesar das diferenças quanto à prescrição de formalidade, foi exposto que XP endereça diversos problemas associados a SOA, identi-

ficados na literatura.

Há diversos métodos ou processos propostos na literatura para implementar um ciclo de desenvolvimento de *software* utilizando SOA, como, por exemplo, o SOMA [53] ou o proposto por DIIRR *et al.* [54]. No entanto, conforme o estudo de mapeamento sistemático conduzido em nossa pesquisa, nenhuma proposta endereça o uso combinado de SOA com métodos ágeis, em geral, ou XP, em particular, para endereçar os problemas expostos.

Além disso, segundo ABRANTES & TRAVASSOS [21], conhecer o contexto de uma organização é fundamental para o sucesso da adoção de métodos ágeis na resolução dos problemas mapeados. Organizações que optam pela adoção de métodos ágeis devem ser capazes de identificar quais práticas são efetivamente importantes para o funcionamento de seu processo de *software*, baseadas nas suas características, sua situação atual e nos resultados esperados. Na mesma linha, NASSIR & SAHIBUDDIN [24] afirmam que processos de desenvolvimento adequados são um fator de sucesso para projetos de *software*.

Assim, dado que **(i)** XP endereça diversos problemas de SOA, **(ii)** SOA requer algum nível de formalidade em definição de processos, e **(iii)** processos devem considerar o contexto da organização para favorecer sua adoção bem sucedida, esta dissertação tem por fim propor uma Linha de Processo de *Software*. A finalidade da LPS é possibilitar entregar soluções orientadas a serviço, desenvolvidas por meio de práticas ágeis, com pontos de variabilidade que levem em consideração o contexto de organizações que queiram utilizá-la, a partir das características desta organização e das práticas do XP que esta considerar mais adequadas.

2.4 Estudo de Mapeamento Sistemático

Um dos passos da pesquisa foi verificar a pré-existência de métodos ou processos que endereçassem a construção de soluções orientadas a serviço apoiados por práticas ágeis de uma forma detalhada. Foi buscado um método que definisse papéis, responsabilidades, artefatos e fases através de um estudo de mapeamento sistemático da literatura. O protocolo do estudo encontra-se detalhado no Apêndice 4 6 e é apresentado de forma resumida a seguir.

A pesquisa de trabalhos se baseou na expressão de busca (declarada em sintaxe independente de ferramenta): *(soa or service oriented) AND (xp or "extreme programming"))*.

A busca foi executada sobre os seguintes motores de busca, escolhidos pelo fato de estarem disponíveis no portal da CAPES: Scopus, Compendex, ACM Digital Library, DBLP Complete Search, IEEExplore. Também foram pesquisados anais de conferências, entre os anos de 2005 e 2012, que endereçam trabalhos nas áreas de desenvolvimento orientado a serviço e/ou métodos ágeis, tais como: SBSI, ICEIS, AMCIS, CAiSE, ICIS, ECIS, SOSE, SBCARS, ESELAW e XP Conference. Após baterias de testes e ajustes ao protocolo, foram mapeados 154 artigos para análise. Destes, quatro não estavam disponíveis para *download*, enquanto outros 119 não tinham o *abstract* alinhado ao objetivo da pesquisa. Ao final, 31 artigos restaram para ter suas propostas analisadas.

Durante a análise das propostas, percebeu-se que a maioria oferecia ideias gerais, diretrizes ou mapeamentos entre conceitos ágeis e conceitos de SOA. Além disso, as propostas ou relatos não endereçavam uma forma estruturada de trabalho, ou seja, não definiam um conjunto de papéis, fases, responsabilidades e entregáveis que pudessem ser utilizados por uma equipe que desejasse aplicar a proposta. Da mesma forma, a maioria dos autores não endereçava a fase de construção de serviços, preferindo, com frequência, discutir atividades mais relacionadas ao planejamento e a condução do projeto.

2.4.1Trabalhos relevantes

A análise das 31 propostas finais levou a identificação dos seis trabalhos mais relevantes, resumidos abaixo.

KARSTEN & CANNIZZO [15] relatam o uso de práticas do XP no desenvolvimento de um produto para a British Telecom, onde o time era distribuído geograficamente e funcionalmente. Havia vários times, cada um trabalhando em uma parte do produto maior, e cada um desses times contava com membros dispersos geograficamente. O uso de práticas ágeis tinha por propósito fundamental manter a comunicação fluindo dentro de cada time, e entre os times, além de manter a motivação de todos. Para tal, foram estabelecidos, por exemplo, eventos presenciais onde os membros das equipes se conheciam presencialmente, compartilhavam conhecimentos, trocavam de times e identificavam interdependências e restrições globais. O relato também dá grande ênfase ao uso de testes de aceitação automatizados: (i) para demonstração do produto durante a reunião de entrega; e, (ii) como fonte de documentação para fins de integração entre as partes do produto. Este trabalho foi escolhido como relevante por discutir melhores práticas de comunicação distribuída, a qual é uma configuração constante em times que desenvolvem soluções orientadas a serviço, além de englobar o uso de testes de aceitação.

O trabalho de **LEE *et al.*** [18] foi escolhido por contribuir com boas práticas relaci-

onadas a fase de construção de serviço, cujo tratamento é objetivo desta dissertação. Os autores especificam que projetos de desenvolvimento de serviços devem ter precauções tais como, por exemplo, níveis adequados de granularidade, interoperabilidade, e modo de interação entre serviços, de forma a atender tanto aos requisitos quanto aos princípios de SOA. A proposta de **LEE *et al.*** [18] se junta ao rol dos artigos relevantes por descrever um método para desenvolvimento de serviços, composto de fases, papéis, atividades e entregáveis, que posteriormente é estendido por **TAN *et al.* [14]** para se adaptar aos princípios e práticas de XP. O segundo não introduz nenhum conceito novo, mas oferece mapeamentos, por exemplo, entre as fases definidas por LEE *et al.* [18] e as fases do XP, ou entre os papéis de um e de outro.

O trabalho de **TAN *et al.* [14]**, no entanto, apesar de definir fases, papéis, atividades e entregáveis, e de ter sido proposto para compreender princípios e práticas do XP, apresenta diversos problemas. Por exemplo, apesar do XP ser essencialmente um método iterativo e incremental, a proposta de **TAN *et al.* [14]** não define quais de suas fases são iterativas, com que frequência ocorrem ou quando uma nova iteração se inicia. O trabalho de **TAN *et al.* [14]** tampouco aborda o uso de testes de aceitação automatizados, uma das principais práticas do XP [16]. Analogamente, o uso de codificação guiada por testes, uma prática importante do XP, não é endereçada por **TAN *et al.* [14]**.

A contribuição de **KROGDAHL, LUEF & STEINDL** [10], por sua vez, oferece ideias sobre o uso de práticas ágeis do Lean [42] no desenvolvimento de serviços. Por exemplo, é sugerido o uso de um *backlog* que englobe toda a corporação, priorizado globalmente. Também é proposta que a interface dos serviços seja deixada em aberto e rapidamente colocada em produção, por meio de um piloto funcional, com o objetivo de permitir que a interface e a lógica interna sejam constante e facilmente refatoradas e refinadas para atender às evoluções nas necessidades de negócio. Os autores também sugerem reuniões frequentes entre representantes dos times para sincronização de conhecimento e impedimentos comuns.

O trabalho de **IVANYUKOVICH *et al.*** [55] foi selecionado por delinear os impactos de práticas do XP sobre o desenvolvimento de serviços. Por exemplo, o *Planning Game* pode ser usado para que o time visualize o *backlog* de serviços com um conjunto de funcionalidades a serem desenvolvidas. *Refactoring* pode ser usado para a melhoria contínua dos serviços. O uso de TDD (*Test-Driven Development*) levaria a uma coordenação automática das dependências entre os serviços, tornando mais simples a orquestração de serviços em tempo de execução.

Por fim, o trabalho de **MARANZATO, MARDEN & HERCULANO** [20] contribui

apresentando um *framework* para melhoria de comunicação entre todos os envolvidos no desenvolvimento de um produto, onde times dispersos trabalham em paralelo no desenvolvimento de um produto, característica comum de times SOA. O produto era dividido por funcionalidades, cada qual endereçada por times e PO's distintos. De forma análoga ao proposto por KROGDAHL, LUEF & STEINDL [10], todos os times compartilhavam um "*Mega Backlog*", onde os itens tinham granularidade variável, com o objetivo de permitir aos PO's manter uma visão global do produto. A cada três meses, esse "*Mega Backlog*" era revisitado, para atualizar a visão de todos os envolvidos sobre o produto e facilitar o acordo quanto as prioridades de negócio. Além desse evento, diversos outros são realizados, de forma a manter a comunicação fluindo dentro dos times e entre os times.

A Tabela 2.4 sumariza as características ágeis ou de orientação a serviços endereçadas pelos trabalhos relevantes citados acima, e o nível de detalhe em que são tratados. A última coluna relaciona a característica a uma lacuna identificada na literatura. As lacunas identificadas na literatura serão discutidas na seção seguinte.

Os símbolos indicam o quanto cada característica foi coberta pelos trabalhos relacionados: (-) a característica foi citada, mas não foram apresentados exemplos ou diretrizes de uso; (+) alguns exemplos foram oferecidos, de forma a facilitar a implementação da característica; (++) a característica foi endereçada em nível de detalhe adequado.

Tabela 2.4: Características endereçadas pelos trabalhos relacionados mais relevantes

Característica	**Referência**	**Nível de Detalhe**	**Lacuna**
C1. Frequência de iterações	[20]	+	G1
C2. Testes de aceitação	[14] [15]	- -	G2
C3. TDD	[14] [55]	- -	G3
C4. *Refactoring*	[15] [55]	+ -	G4
C5. Práticas de construção de serviços	[18]	++	G5
C6. Boas práticas para times distribuídos	[20] [15]	++ +	G5

2.4.2Lacunas na literatura

As seguintes lacunas na literatura foram identificadas, baseado no princípios de SOA, nas características de XP e análise dos trabalhos identificados no estudo de mapeamento sistemático.

G1. Iterações: A maioria dos autores apresenta ideias sobre o uso de práticas do XP no desenvolvimento de serviços, mas não discutem o uso de iterações em um modelo de desenvolvimento de serviços, apesar de XP ser, por definição, um método iterativo e incremental [56]. O ciclo de vida do XP é dividido em cinco fases iterativas [16]. O método proposto por MARANZATO, MARDEN & HERCULANO [20] endereça com riqueza de detalhes o uso de iterações, ao passo que o método proposto por TAN *et al.* [14] inclui fases, atividades, papéis e entregáveis, mas não define quão iterativa é cada fase, nem em que momento a iteração é reiniciada, por isso não figura na Tabela 2.4.

G2. Testes de aceitação: Nenhum dos autores discute em profundidade o uso de testes de aceitação, apesar da sua importância no ciclo do XP. É fundamental para a aderência do código de produção aos testes de aceitação para determinar se a funcionalidade está finalizada [16] [56]. A única proposta que o aborda, ainda que de forma superficial, é a de TAN *et al.* [14]. Esta proposta discute o uso de critérios de aceitação, escritos pelo PO com a assistência do analista de testes, mas estes não são transformados e automatizados como testes de aceitação, ficando restritos à execução manual em algum momento não definido no método. O trabalho de KARSTEN & CANNIZZO [15] faz meramente um relato sobre o uso de testes de aceitação automatizados em um time distribuído, sem propor uma forma estruturada de se trabalhar com iterações.

G3. Testes unitários: Apesar da importância dos testes serem escritos antes do código de produção [16], nenhum dos autores menciona o uso de testes unitários durante o desenvolvimento dos serviços, nem discute melhores práticas ou problemas inerentes ao desenvolvimento de serviços guiado por testes. TAN *et al.* [14] menciona o uso de testes unitários num cenário onde o serviço instalado em um ambiente é a unidade, ao passo em que no XP [16] o teste unitário é construído em conjunto com cada unidade de código (ex.: um teste unitário para cada classe do sistema). IVANYUKOVICH *et al.* [55], por sua vez, apenas menciona que testes unitários podem ser usados para diminuição do acoplamento de um serviço, sem oferecer diretrizes de uso de testes unitários no desenvolvimento de serviços.

G4. *Refactoring*: Apesar de ser uma prática rotineira do XP [16] [56], *Refactoring* também não é discutido em detalhe pelos autores mais relevantes. Não há discussões

acerca de, por exemplo, impactos decorrentes de refatoração de interfaces sobre consumidores e parceiros de negócio.

G5. Boas práticas relacionadas a construção de serviços: A maioria dos trabalhos pesquisados não menciona diferenças ou problemas inerentes ao desenvolvimento de serviços, apesar de haver diversas diferenças catalogadas entre a engenharia de *software* orientada a serviços e a engenharia de *software* dita "tradicional" [8], especialmente no que se refere a questões associadas a comunicação em times distribuídos, configuração frequente em projetos de desenvolvimento de serviços. As exceções são os trabalhos de MARANZATO, MARDEN & HERCULANO [20], TAN *et al.* [18] e KARSTEN & CANNIZZO [15], os quais endereçam questões relacionadas a construção de serviços e/ou times distribuídos.

3. LPS para Soluções Orientadas a Serviço

Conforme apresentado na Seção 1.2, o objetivo desta dissertação é propor uma Linha de Processo de *Software* (LPS), cujo objetivo é entregar soluções orientadas a serviço. Soluções orientadas a serviço acarretam uma série de questões, as quais serão tratadas por meio de práticas ágeis do XP na LPS proposta. A LPS visa endereçar também questões relativas ao contexto de organizações, permitindo variações em função de suas características e necessidades.

O foco da LPS é exclusivamente a fase de construção de um ciclo de desenvolvimento de *software*. Esta proposta endereça os princípios de SOA e lacunas na literatura catalogadas no Capítulo 2.

Para a definição desta proposta, foi utilizada a abordagem *top-down*, proposta por BARRETO [48], e apresentada na Seção 3.3. Após esta seção, são apresentados os passos da abordagem de BARRETO seguidos na definição da Linha de Processo. Em seguida, são apresentados os resultados obtidos e não-conformidades encontradas na avaliação da LPS proposta por especialistas nos assuntos de XP, SOA e componentização de processos. No entanto primeiramente serão introduzidos os principais conceitos ágeis referenciados no conteúdo da LPS.

3.1Conceitos ágeis utilizados na definição da LPS

Esta seção descreve os principais conceitos ágeis empregados na definição da LPS, e que são importantes para o entendimento do trabalho. XP compartilha diversas características e práticas com outros métodos ágeis [17], em especial com o Scrum. Assim, algumas das práticas apresentadas abaixo têm sua definição baseada na literatura disponível sobre o Scrum.

3.1.1 *Product Backlog*

Segundo o SCRUM GUIDE [57], o *Product Backlog* corresponde a uma lista de todas as funcionalidades a serem implementadas ao longo de um projeto. SCHWABER & BEEDLE [58] complementam essa definição apresentando que o *Product Backlog* corresponde a uma lista de requisitos funcionais e não-funcionais, a qual, quando traduzida em funcionalidades para o usuário final, concretizarão a visão do sistema. Tipicamente, cada elemento da lista será composto de uma descrição sucinta de seu propósito, sua medida de prioridade para o negócio e sua medida de complexidade de implementação, conforme o exemplo apresentado na Tabela 3.1.

Tabela 3.1: Exemplo de *Product Backlog* (adaptado de [57])

ID	***User Story***	**Estimativa**	**Prioridade**
7	EU COMO um usuário autorizado DESEJO abrir uma nova conta	3	1
1	EU COMO um usuário não-autorizado DESEJO entrar no sistema	1	2
10	EU COMO um usuário autorizado DESEJO sair do sistema	1	4
9	Criar script pra limpar base de dados	1	4
2	EU COMO um usuário autorizado DESEJO visualizar a lista de itens A FIM DE escolher um item para trabalhar	2	5

Apesar do termo *Product Backlog* ser formalmente definido na literatura relativa ao Scrum, o XP também lança mão de uma lista de itens priorizados para planejamento de iterações e *Releases* [56].

De fato, segundo COHN [59], do ponto de vista do *Product Backlog*, a única diferença entre o XP e o Scrum é o fato de no XP, a equipe implementar os itens na sequência de itens priorizada pelo cliente (apesar desse conjunto de itens não ter um nome próprio). Já no Scrum, a equipe decide a sequência em que os itens serão implementados, baseados, entre outros fatores, na priorização dos mesmos.

Assim, mesmo não sendo um termo definido formalmente na literatura do XP, usaremos o termo *Product Backlog* para denominar o conjunto de funcionalidades a serem implementados ao longo de um projeto.

3.1.2 *User Stories*

Conforme descrito na Seção 3.1.1, cada item que compõe o *Product Backlog* é um requisito funcional ou não-funcional. Tais requisitos podem ser detalhados em formatos diversos, como, por exemplo, casos de uso ou *User Stories*, formato definido no XP [60].

Uma *User Story* representa a visão do usuário do sistema sobre o comportamento do sistema, descrita de forma resumida e sucinta [60]. Segundo COHN [61], uma *User Story* descreve uma funcionalidade que agregará valor a um usuário ou a um comprador de um sistema, sendo composta de:

- Uma descrição sucinta e por escrito da funcionalidade, usada não como documentação ou representação da mesma, mas apenas com o propósito de lembrete do que deve ser feito, para fins de planejamento e priorização do *Backlog*;
- Conversas sobre a funcionalidade, a fim de levantar a discussão sobre seus detalhes;
- Testes que documentam detalhes e podem ser usados para determinar quando uma *User Story* está completa.

Tipicamente, as descrições das *User Stories* são escritas a mão, em cartões de papel. No entanto, apesar de serem a manifestação mais palpável de uma funcionalidade, não são as mais importantes, papel esse ocupado pelos testes de aceitação.

Não há um formato definido para uma *User Story*. Tome-se, por exemplo, o desenvolvimento de um sistema de cadastro e busca de vagas de emprego. A descrição de uma *User Story* poderia ser tão simples quanto “Um usuário deve poder procurar por empregos”ou ”Uma empresa deve poder cadastrar sua oferta de vagas”.

Um dos formatos mais difundidos na indústria é o “Role-Feature-Reason”, também

conhecido como formato *Connextra*, onde os requisitos são representados por meio de três expressões-chave:

- **Eu Como**: indica o papel do usuário no contexto de negócio;
- **Desejo**: indica a ação ou funcionalidade a ser utilizada pelo usuário no sistema;
- **A Fim De**: indica o objetivo de negócio a ser atendida pela funcionalidade.

Um exemplo de uma *User Story*, no mesmo contexto de um sistema de oferta de empregos, poderia ser:

- **Eu Como**: gerente de RH da empresa XYZ;
- **Desejo**: cadastrar as vagas disponíveis em minha empresa;
- **A Fim De**: poder atender ao volume de demandas requisitado por nossos clientes.

3.1.3 *Critérios de Aceitação*

Na Seção 3.1.2, foi descrito que os requisitos funcionais e não-funcionais de um projeto XP tipicamente usam o formato de *User Stories*, o qual preconiza três passos: escrita no cartão, conversa e testes.

Os critérios de aceitação compõem esse terceiro passo, servindo para definir os passos de verificação da completude de uma *User Story*. Segundo COHN [61], os critérios de aceitação (também denominados "testes de aceitação") definem o processo pelo qual as *User Stories* serão validadas, verificando que as mesmas foram desenvolvidas de tal forma que atendem exatamente ao esperado pelo cliente.

Aqui também não há a obrigação de um formato pré-definido. Tomando como exemplo novamente o sistema de cadastro de vagas, num cenário onde uma empresa pague para expor suas vagas, uma funcionalidade de pagamento pelo cadastro poderia ter os seguintes critérios de aceitação:

- Deve ser possível pagar com Visa, MasterCard e Amex;
- Não deve ser possível pagar com Diner's;
- Não deve ser possível pagar com um cartão expirado.

Um formato usado para critérios de aceitação bastante difundido é o *Given-When-Then*[1]:

- **Dado**: contexto do critério;
- **Quando**: indica a ação (ou evento) executada pelo usuário;
- **Então**: consequências da ação;

Exemplos de aplicação deste formato, considerando o sistema de exemplo descrito nas *User Stories*, poderiam ser:

- **Dado**: que a vaga "Analista de Sistemas Pleno" foi totalmente descrita e está pronta para pagamento;
- **Quando**: o gerente de RH informa os dados de seu cartão de crédito Diner's;
- **Então**: o sistema informa que cartões Diner's não são permitidos;

- **Dado**: que a vaga "Analista de Sistemas Pleno" foi totalmente descrita e está pronta para pagamento;
- **Quando**: o gerente de RH informa os dados de seu cartão de crédito Visa;
- **Então**: o sistema informa que o pagamento foi aceito e envia e-mail de confirmação para a empresa XYZ;

É importante ressaltar que os critérios de aceitação detalham a funcionalidade do ponto de vista de negócio, independente de mídia ou tecnologia. Não há referências a eventos de tela, cliques ou outros detalhes tecnológicos.

3.1.4 *Story Points*

Segundo COHN [61], uma abordagem ótima para estimativa de *User Stories* deveria:

- permitir mudanças, dadas novas informações sobre as *User Stories*;
- funcionar independente da granularidade;

[1] `http://guide.agilealliance.org/guide/gwt.html`

- tomar pouco tempo;
- informar rapidamente sobre a quantidade de trabalho que resta ser feita;
- ser tolerante à imprecisão;
- poder ser usada para planejar *Releases*.

Uma técnica que se adequa a estes requisitos é a denominada “Story Points”, segundo a qual o time define qual será sua unidade de estimativa, como, por exemplo, um dia de trabalho sem interrupções ou uma semana de trabalho sem interrupções [61]. O importante é se tratar de uma medida abstrata e imprecisa de forma que times distintos não possam ser comparados pela mesma métrica (afinal, lidam com complexidades e problemas distintos) e que as estimativas possam acomodar uma variação positiva ou negativa, dentro de uma projeção aproximada.

3.1.5*Product Owner*

Segundo o SCRUM GUIDE [57], o *Product Owner* (PO) é a pessoa responsável por gerenciar o *Product Backlog* e maximizar tanto o valor do produto sendo desenvolvido quanto o valor do time. O PO é responsável por definir e priorizar os itens do *Backlog*, garantindo sua visibilidade e entendimento por todos os envolvidos.

O PO pode representar os interesses de diversos clientes sobre um produto. Neste caso, ele é responsável por coordenar seus interesses e chegar a um acordo sobre a melhor priorização do ponto de vista da organização como um todo.

3.1.6*Sprint*

De acordo com o SCRUM GUIDE [57], uma *Sprint*, ou iteração, é um período fixo de tempo, com idealmente um mês ou menos de duração, onde, ao seu final, é criado um incremento ao produto que segue o conceito de “Pronto”.

SCRUM GUIDE [57], “Pronto” é um acordo estabelecido entre os membros do time e o PO. Por exemplo, um dado incremento ao produto pode ser considerado “Pronto” se, e somente se, o incremento pode ser imediatamente aplicado em produção. Por outro lado, um time em outro contexto pode convencionar que um incremento esteja “Pronto” a partir do momento em que possa ser testado.

As *Sprints* têm um objetivo bem definido, ou seja, cada *Sprint* tem um conjunto definido de funcionalidades a serem entregues dentro do que é definido como “Pronto”, além

de um planejamento flexível de como entregá-las dentro do tempo planejado.

3.1.7*Release*

No escopo deste trabalho, uma *Release* será definida como um conjunto de *Sprints*, com um objetivo bem definido. Ou seja, ao fazer o planejamento de uma *Release*, esta definirá o escopo a ser atendido, bem como o número de *Sprints* inicialmente necessárias para atingi-lo.

3.2Pré-condições ao uso da LPS

As pré-condições para uso da LPS são:

- *Backlog* elaborado, ou seja, listagem das funcionalidades a serem desenvolvidas ao longo da *Release*;
- As funcionalidades do *Backlog* devem estar estimadas, por exemplo, através de *Story Points* como proposto por COHN [61], a fim de auxiliar no acompanhamento do andamento da iteração;
- Os requisitos destas funcionalidades devem ter sido previamente levantados, estando descritos por meio de *User Stories* e critérios de aceitação;
- O conceito de "Pronto" deve ter sido previamente acordado entre PO e equipe de desenvolvimento.

Antes do início da iteração ou *Release*, o PO é responsável por criar e/ou manter atualizado o *Backlog* com as funcionalidades a serem desenvolvidas ao longo do projeto. Este *Backlog* deve estar disponível em local acessível a qualquer momento pela equipe de desenvolvimento e envolvidos, para acompanhamento das mudanças nas prioridades do negócio. A qualquer momento, este *Backlog* conterá funcionalidades estimadas por meio de *Story Points*. Em um primeiro momento, na ausência de requisitos mais refinados e à luz do entendimento inicial das necessidades do cliente, estas estimativas serão feitas em alto nível, podendo variar com o tempo, conforme, a cada iteração, os requisitos de granularidade mais fina são apresentados, avaliados e entendidos pela equipe de desenvolvimento.

Antes da reunião de planejamento da iteração (*Iteration Planning*), com antecedência acordada entre PO e equipe de desenvolvimento, o PO fica responsável por disponibilizar

à equipe de desenvolvimento os requisitos das funcionalidades priorizadas para serem desenvolvidas na próxima iteração. A equipe, por sua vez, é responsável por validar estes requisitos e verificar se tem capacidade de entregar todo o escopo priorizado ao final da *Sprint*. Desta forma, o escopo da *Sprint* é definido através de um acordo entre PO e a equipe de desenvolvimento.

Os requisitos devem estar descritos por meio de *User Stories* e critérios de aceitação, através de formato acordado entre todos os envolvidos, de forma a favorecer uma linguagem comum a todos. Os requisitos devem preferencialmente ser disponibilizados por meio de ferramenta que permita verificar as diferentes versões geradas ao longo do tempo e a qual todos tenham acesso a qualquer momento, como, por exemplo, ferramentas wiki ou de gerência de configuração.

3.3Definição das características de processo

Conforme definido por BARRETO [48], a estratégia *top-down* é guiada, primeiramente, pela definição das características a serem atendidas pelo processo. Em seguida, a partir destas características, são derivados componentes de processo e atividades as quais, em última instância, definirão a LPS.

As características definidas para a LPS se apoiaram em diversas fontes:

- Os princípios de SOA, conforme definido por ERL [4];
- As práticas de XP, segundo BECK [16];
- Melhores práticas propostas por autores diversos nas áreas de métodos ágeis e SOA;
- Lacunas na literatura apresentadas por AZEVEDO e SANTOS [25], a partir de um estudo de mapeamento sistemático, foram tratadas.

O conjunto final de características de processo é apresentado na Tabela 3.2.

Tabela 3.2: Características de processo para a Linha de Processo de *Software* para Soluções Orientadas a Serviço

Requisito	**Característica de processo**
Deve haver uma cerimônia de início de cada sprint, onde os requisitos serão apresentados, segundo a prática *Iteration Planning Meeting* do XP [16]	*Iteration Planning*
Deve atender o uso de testes de aceitação, os quais, segundo BECK [16], são o ponto de partida de qualquer atividade de codificação no XP, segundo a prática *Testing* do XP	Testes de aceitação
As atividades de codificação e teste devem ser atendidas em conjunto, com os testes escritos antes do código de produção, conforme preconiza a prática *Testing* do XP [16]	TDD
As atividades de codificação e teste devem ser executadas em pares, conforme preconiza a prática *Pair Programming* do XP [16]	Programação em Pares
Continua na próxima página	

Tabela 3.2 – continuação da página anterior

Requisito	Característica de processo
Todo teste deve ser automatizado e incorporado a suíte de testes, provendo *feedback* sobre a robustez da solução a cada integração do código, conforme a prática *Continuous Integration* do XP [16]	Integração Contínua
Deve haver uma cerimônia de demonstração de cada *Sprint*, segundo a prática *Iteration Demonstration Meeting* do XP [62]	*Iteration Demonstration*
Deve procurar maximizar o reuso, para ser aderente ao princípio *Service Reusability* [4]	Reuso
Deve procurar maximizar a expressividade dos contratos dos serviços através de padrões pré-estabelecidos, de forma a ser aderente ao princípio *Standardized Service Contract* [4]	Contrato
Deve minimizar a dependência entre contrato, consumidor e lógica interna, de forma a ser aderente ao princípio *Service Loose Coupling* [4]	Baixo Acoplamento
Todo teste deve ser automatizado e incorporado a suíte de testes, provendo	Integração Contínua
Continua na próxima página	

Tabela 3.2 – continuação da página anterior

Requisito	Característica de processo
feedback sobre a robustez da solução a cada integração do código, conforme a prática *Continuous Integration* do XP [16]	
Deve minimizar a dependência entre o serviço e seu ambiente de execução, de forma a ser aderente ao princípio *Service Autonomy* [4]	Autonomia
Deve minimizar a quantidade de informações de estado gerenciadas pelo serviço, de forma a ser aderente ao princípio *Service Statelessness* [4]	Ausência de estado
Deve maximizar a geração de informações que facilitem a descoberta dos serviços, de forma a ser aderente ao princípio *Service Discoverability* [4]	Descoberta
Deve minimizar a dependência entre contrato, consumidor e lógica interna do serviço, de forma a ser aderente ao princípio *Service Loose Coupling* [4]	Baixo Acoplamento

Continua na próxima página

Tabela 3.2 – continuação da página anterior

Requisito	Característica de processo
Deve atender a comunicação entre membros de equipes distribuídas [15] para ser aderente a característica de SOA de lidar com equipes dispersas geograficamente	Equipes distribuídas
Deve atender ao desafio de SOA "refatoração de contratos", relacionado ao desafio "distribuição de serviços através das fronteiras organizacionais" [6], sem perder aderência aos princípios *Standardized Service Contract* e *Service Discoverability*	Modificação de contrato
Deve atender ao desafio *testing to account for the multiple systems interoperability* [14] para ser aderente ao princípio *Service Composability* [4] e à prática *Testing* do XP [16]	Testes distribuídos
Deve atender boas práticas que minimizem problemas de interoperabilidade entre tecnologias, de forma a tratar o desafio "distribuição de serviços através das fronteiras organizacionais" [6] e o fundamento de interoperabilidade entre serviços [4]	Distribuição de serviços

Continua na próxima página

Tabela 3.2 – continuação da página anterior

Requisito	Característica de processo
Deve considerar organizações onde o PO não tenha disponibilidade para fornecer os requisitos da iteração à equipe. De acordo com MELO *et al.* [44], é comum que haja indisponibilidade de parte da equipe de produto para seguir as práticas preconizadas pelos métodos ágeis, sendo, assim, necessário considerar situações onde o PO não esteja disponível para fornecer os requisitos da iteração à equipe.	Fornecimento de requisitos na ausência do PO
Deve considerar organizações onde o PO não tenha disponibilidade para fornecer informações de potencial de reuso à equipe. De forma análoga a característica anterior, dado que é comum haver indisponibilidade de parte da equipe de produto, é necessário endereçar situações onde o PO não esteja disponível para fornecer informações de potencial de reuso a equipe.	Fornecimento de informações de potencial de reuso na ausência do PO
Continua na próxima página	

Tabela 3.2 – continuação da página anterior

Requisito	Característica de processo
Deve considerar organizações cujo ferramental não inclua um barramento de serviços. Idealmente, testes de serviços deveriam seguir as orientações de RIBAROV *et al.* [75]. No entanto, dependendo do tamanho e condições econômicas da organização, pode não ser possível arcar com os custos de implantação de um barramento de serviços. Assim, esta característica do contexto de uma organização deve ser endereçada de forma que os testes não sejam prejudicados, o que traria impactos a qualidade da solução final.	Testes de integração sem barramento
Deve considerar situações onde um teste de integração entre serviços seja inviável, sob a ótica econômica ou de inviabilidade técnica. Por exemplo, se um dado serviço, ao ser testado, faz uma invocação a um parceiro de negócios e esta invocação é cobrada, então pode ser economicamente inviável para a organização lidar com este tipo de testes de integração.	Testes de integração sem invocações reais
	Continua na próxima página

Tabela 3.2 – continuação da página anterior

Requisito	Característica de processo
Deve considerar organizações em que não haja disponível ferramental para automação da disponibilização do serviço. Idealmente, o serviço deveria ser instalado automaticamente, ao fim do ciclo de integração contínua, conforme as práticas do XP. No entanto, a aquisição de tal ferramental pode ser provar economicamente inviável a uma organização. Desta forma, esta característica de organizações deve ser endereçada.	Disponibilização de serviço de forma manual

O estudo de mapeamento sistemático apresentado por CARVALHO, AZEVEDO & SANTOS [25] identifica cinco lacunas na literatura disponível no que se refere ao uso de métodos ágeis para a construção de soluções orientadas a serviço. Estas lacunas ressaltam a necessidade que pesquisas sejam realizadas para:

- Tratamento de iterações num ciclo de desenvolvimento orientado a serviços;
- Melhores práticas de uso de testes de aceitação, prática fundamental de XP, no ciclo de desenvolvimento de soluções orientadas a serviço;
- Uso de testes automatizados em paralelo à codificação;
- Melhores práticas de refatoração de serviços, especialmente no que se refere a contratos;
- Incorporação dos princípios de orientação a serviços às preocupações inerentes ao *design* de serviços;

- Endereçamento de práticas para times distribuídos, característica fundamental no desenvolvimento de soluções orientadas a serviço;

Estas lacunas criaram inicialmente a necessidade pelas características **Testes de aceitação**, **TDD**, **Equipes distribuídas**, **Modificação de contrato**. Outras características derivaram dos princípios de SOA, devido a lacuna que foi identificada de baixa correlação entre os trabalhos analisados e estes princípios. Assim, foram incorporadas as características **Reuso**, **Contrato**, **Baixo Acoplamento**, **Abstração**, **Autonomia**, **Ausência de Estado**, **Descoberta** e **Composabilidade**.

As lacunas relativas a boas práticas de desenvolvimento orientado a serviços e times distribuídos, por sua vez, foram expandidas, derivando as características **Equipes distribuídas** e **Distribuição de serviços**.

As práticas e princípios de XP, em combinação a práticas e princípios de SOA, levaram a incorporação de outras características. Por exemplo, a lacuna relativa a iterações levou a criação de características que tratassem o início e fim de iterações, sendo estas as características **Iteration Planning** e **Iteration Demonstration**. As características **TDD**, **Programação em Pares**, **Integração Contínua** foram criadas a partir das práticas *Testing*, *Pair Programming* e *Continuous Integration* do XP.

Por fim, uma característica foi adicionada a partir do cruzamento de lacunas. A característica de **Testes distribuídos** foi influenciada pela prática *Testing* do XP em conjunto ao princípio de *Service Composability* e ao desafio de testes distribuídos proposto por TAN *et al.* [14].

Ao final da identificação das características, não foi identificada nenhuma dependência ou conflito entre as mesmas.

3.4Definição e caracterização dos elementos de processo

A segunda etapa da estratégia *top-down* [48] diz respeito a definição e caracterização dos componentes de processo que atenderão as características delineadas no passo anterior e, em última instância, irão compor a Linha de Processo de *Software*. Parte deste passo diz respeito também a identificação das partes comuns e pontos de variação previstas para os processos instanciados a partir da LPS definida.

Estes pontos em comum e variantes são modelados através de componentes concretos, componentes abstratos e atividades. Os componentes abstratos identificam pontos de

variação observados em um comportamento. As partes invariantes de um comportamento são definidas por meio de um componente concreto. Caso sejam identificadas variantes e seja necessário o uso de um componente abstrato, os componentes concretos que o especializam devem ser definidos. O reverso não é verdadeiro, ou seja, a existência de um componente concreto não depende de um componente abstrato. Quando for observada em um componente concreto a existência de partes menores constantes não reutilizáveis, estas partes menores serão representadas por meio de atividades.

A definição dos componentes e atividades foi feita tomando por base *templates* de definição de processos do grupo de Qualidade de *Software* da COPPE/UFRJ [52] [23]. Estes *templates* foram customizados para atender aos propósitos desta dissertação. As tabelas abaixo apresentam os *templates* utilizados.

Tabela 3.3: *Template* utilizado para definição dos componentes de processo

Identificador	**<Identificador único do componente de processo >**
Nome:	<Nome do componente>
Descrição:	<Descrição do componente>
Tipo de Componente:	<Concreto/Abstrato>
Mandatório:	<Indica se o componente é de execução obrigatória>
Participantes:	<Papéis que atuam como interessados na execução do componente>
Responsáveis:	<Papéis que executam o componente>
Critérios de Entrada:	<Pré-condições à execução do componente>
Critérios de Saída:	<Pós-condições à execução do componente>
Artefatos Requeridos:	<Artefatos necessários a execução do componente>
Artefatos Produzidos:	<Artefatos produzidos durante a execução do componente>
Características Atendidas:	<Características endereçadas pelo componente>
Variantes deste Componente:	<Identificadores dos componentes variantes (aplicável somente a componentes abstratos)>
Arquitetura Interna:	<Componentes e/ou atividades que compõem a estrutura interna do componente, caso aplicável>

O campo “Identificador”, utilizado no *template* de definição de componentes, é regido

Tabela 3.4: *Template* utilizado para definição de atividades

Atividade	**<Nome da atividade>**
Descrição:	<Descrição do componente>
Participantes:	<Papéis que atuam como interessados na execução da atividade>
Responsáveis:	<Papéis que desempenham a atividade>
Artefatos Requeridos:	<Artefatos necessários a execução da atividade>
Artefatos Produzidos:	<Artefatos produzidos durante a execução da atividade>

pela lei de formação a seguir. Tome-se como exemplo o componente com o identificador UNR.AVRE.CONC.0001:

- UNR: refere-se à organização que define o componente (UNIRIO);
- AVRE: sigla que identifica o processo (Avaliar requisitos);
- CONC: indica se o componente é concreto (CONC) ou abstrato (ABS);
- 0001: número identificador;

Os seguintes trabalhos foram utilizados como base para a definição dos componentes e atividades:

- ***eXtreme Programming Explained*** [16]: Literatura básica sobre XP, este livro apresenta em profundidade os princípios, valores e práticas deste método ágil;
- ***SOA - Principles of Service Design***: Este livro define em detalhes os princípios de orientação a serviços, além de apresentar os requisitos a serem atendidos para que uma solução seja aderente aos mesmos;
- **CARVALHO, AZEVEDO & SANTOS [25]**: Este trabalho apresenta um estudo de mapeamento sistemático onde são identificadas as lacunas na literatura que esta dissertação se propõe a tratar.

Para atender aos requisitos delineados na Seção 3.3, foram definidos 41 componentes, em conjunto com as atividades que formam as respectivas arquiteturas internas. A fim

de exemplificar a execução deste passo da abordagem *top-down*, são apresentados a seguir, exemplos de cada tipo de componente: um componente abstrato com seus variantes concretos (Tabela 3.5); um componente concreto sem arquitetura interna (Tabela 3.6); e, um componente concreto com arquitetura interna (Tabela 3.7). A relação de todos os componentes consta no apêndice III desta dissertação.

Tabela 3.5: Exemplo de componente concreto e seus variantes concretos

Identificador	**UNR.FORE.ABST.0005**
Nome:	Fornecer requisitos da iteração à equipe
Descrição:	Este componente visa fornecer as *User Stories* e critérios de aceitação das funcionalidades a serem desenvolvidas na interação. Isto pode ser feito de formas variadas, conforme as variantes deste componente. No entanto, independente da forma escolhida, este componente envolve: • Fornecer as *User Stories* que foram priorizadas, em conjunto com seu valor e os objetivos de negócio relacionados; • Fornecer os critérios de aceitação de cada funcionalidade, ilustrando os cenários de negócio a serem atendidos; Eventualmente haverá dúvidas ou pontos a serem esclarecidos acerca dos requisitos. Estas dúvidas devem ser registradas para posterior esclarecimento

	ou devem ser esclarecidas imediatamente pelo PO, de acordo com a variante concreta escolhida.
Tipo de Componente:	Abstrato
Mandatório:	Sim
Participantes:	• Equipe de desenvolvimento
Responsáveis:	• PO
Critérios de Entrada:	• Requisitos aceitos pela equipe de desenvolvimento • *Backlog* da iteração definido
Critérios de Saída:	• Requisitos fornecidos
Artefatos Requeridos:	• *Backlog* de funcionalidades • Requisitos
Artefatos Produzidos:	Não há
Características Atendidas:	• *Iteration Planning*
Variantes deste Componente:	• UNR.APRE.CONC.0006 • UNR.DIRE.CONC.0007
Arquitetura Interna:	Não há
Identificador	**UNR.APRE.CONC.0006**
Nome:	Apresentar requisitos da iteração à equipe
Descrição:	Nesta variante, o fornecimento dos requisitos se dá durante a reunião de planejamento da iteração (*Iteration planning meeting*), onde o PO fica responsável por apresentar a toda a equipe de desenvolvimento os requisitos das funcionalidades que foram priorizadas e validadas

	para serem trabalhadas ao longo da iteração que se inicia. Nesta variante, o PO fica responsável por providenciar os recursos e ferramentas necessários à execução do componente, de acordo com as características da equipe (co-localizada ou distribuída). Exemplos de recursos incluem salas de reunião, ferramentas de áudio/videoconferência, projetores, *etc.*). Ao longo da reunião, o PO apresenta as *User Stories*, explicando seu valor e os objetivos de negócio que serão alcançados com a entrega daquela funcionalidade. Os critérios de aceitação também são apresentados, de forma a deixar a equipe ciente dos cenários de negócio a serem atendidos pela funcionalidade, e esclarecendo eventuais dúvidas. Nesta variante, se houver dúvidas ou pontos a serem esclarecidos, estes podem ser registrados em ferramenta previamente acordada ou esclarecidos imediatamente, dependendo da complexidade destes pontos.
Tipo de Componente:	Concreto

Mandatório:	Não
Participantes:	• Equipe de desenvolvimento
Responsáveis:	• PO
Critérios de Entrada:	• Requisitos aceitos pela equipe de desenvolvimento • *Backlog* da iteração definido
Critérios de Saída:	• Requisitos fornecidos
Artefatos Requeridos:	• *Backlog* de funcionalidades • Requisitos
Artefatos Produzidos:	Não há
Características Atendidas:	• *Iteration Planning*
Variantes deste Componente:	Não há
Arquitetura Interna:	Não há
Identificador	**UNR.DIRE.CONC.0007**
Nome:	Disponibilizar requisitos da iteração à equipe sem apresentação
Descrição:	Nesta variante, o fornecimento dos requisitos se dá por meio de ferramenta previamente acordada, não havendo apresentação por parte do PO. Mediante a disponibilização destas informações, a equipe de desenvolvimento fica responsável por registrar dúvidas e pontos a serem esclarecidos posteriormente pelo PO.
Tipo de Componente:	Concreto

Mandatório:	Não
Participantes:	• Equipe de desenvolvimento
Responsáveis:	• PO
Critérios de Entrada:	• Requisitos aceitos pela equipe de desenvolvimento • *Backlog* da iteração definido
Critérios de Saída:	• Requisitos fornecidos
Artefatos Requeridos:	• *Backlog* de funcionalidades • Requisitos
Artefatos Produzidos:	Não há
Características Atendidas:	• *Iteration Planning*
Variantes deste Componente:	Não há
Arquitetura Interna:	Não há

Tabela 3.6: Exemplo de componente concreto sem arquitetura interna

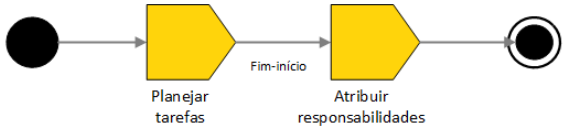
Identificador	**UNR.ESRE.CONC.0011**
Nome:	Esclarecer requisitos
Descrição:	Nesta variante, o fornecimento dos requisitos se dá Caso os requisitos fornecidos tenham dado margem a dúvidas ou necessidade esclarecimento posterior, o PO pode escolher esclarecê-los durante a apresentação dos mesmos e/ou esclarecê-los após seu registro em ferramenta previamente acordada.

	Neste componente, havendo pontos pendentes de esclarecimento via ferramenta, os esclarecimentos devem ser também registrados, preferencialmente na mesma ferramenta, de forma que dúvidas e respectivos esclarecimentos sejam facilmente identificáveis e relacionados. No caso de esclarecimentos por meio de ferramenta, se a mesma não der suporte a notificações automáticas, o PO fica responsável por notificar a equipe de desenvolvimento que os esclarecimentos foram registrados.
Tipo de Componente:	Concreto
Mandatório:	Não
Participantes:	• Equipe de desenvolvimento
Responsáveis:	• PO
Critérios de Entrada:	• Há pontos pendentes de esclarecimentos
Critérios de Saída:	• Pontos esclarecidos
Artefatos Requeridos:	• Pontos pendentes de esclarecimentos
Artefatos Produzidos:	Esclarecimento das dúvidas
Características Atendidas:	• *Iteration Planning*
Variantes deste Componente:	Não há
Arquitetura Interna:	Não há

Tabela 3.7: Exemplo de componente concreto com arquitetura interna

Identificador	**UNR.DERE.CONC.0023**
Nome:	Definir responsáveis pelas tarefas
Descrição:	O propósito deste componente é definir que pessoas, dentro da equipe de desenvolvimento, serão responsáveis por quais tarefas durante o desenvolvimento da funcionalidade. Complementarmente, este componente visa definir todas as tarefas a serem executadas pela equipe, ao longo do seu desenvolvimento, até que a funcionalidade seja considerada completa e pronta para validação pelo PO, conforme representado na Seção **Arquitetura Interna** a seguir. Ao fazer o planejamento de tarefas e responsáveis, a equipe fica responsável por decidir o ferramental necessário à execução da tarefa, de acordo com sua configuração (co-localizada ou distribuída). Assim como nos componentes anteriores, este ferramental pode incluir ferramentas de áudio/videoconferência, salas de reunião, registro do planejamento, entre outras.

	De forma a permitir o acompanhamento instantâneo do desenvolvimento por todos os envolvidos, equipes distribuídas devem preferencialmente registrar seu progresso em meio digital e online. Este componente é executado ao início de cada funcionalidade planejada para a iteração. Ao final deste componente, o planejamento de tarefas de cada funcionalidade priorizada para a iteração é registrado, em conjunto com os responsáveis pelas primeiras tarefas.
Tipo de Componente:	Concreto
Mandatório:	Sim
Participantes:	• Equipe de desenvolvimento
Responsáveis:	• Equipe de desenvolvimento
Critérios de Entrada:	• Formato dos testes de integração definido • Estratégia de codificação de testes definida
Critérios de Saída:	• Responsáveis atribuídos às tarefas
Artefatos Requeridos:	• *Backlog* da iteração • Requisitos • Contratos dos serviços
Artefatos Produzidos:	• Planejamento de tarefas a serem executadas até a conclusão de cada funcionalidade, com

	respectivos responsáveis, se aplicável.
Características Atendidas:	• Equipes distribuídas
Variantes deste Componente:	Não há
Arquitetura Interna:	

Os componentes construídos nesta LPS foram representados por meio da notação proposta por BARRETO [48] e apresentada na Tabela 2.3. A ferramenta para representação dos componentes e linhas com esta notação gráfica está disponível no Ambiente de Alta Maturidade (A2M) desenvolvido na COPPE/UFRJ.

3.5Estruturação e caracterização da linha de processo

A terceira etapa da estratégia *top-down* [48] se refere a estruturação e caracterização da LPS. Isto se atinge determinando a forma como os elementos de processo se conectam e a obrigatoriedade de execução de cada componente, bem como pela verificação do encadeamento dos artefatos entre os componentes e pela verificação de quão alcançáveis são os critérios de entrada e saída de cada componente. Por fim, as características do processo devem também ser atreladas aos componentes aplicáveis, explicitando os pontos de variação da LPS, e as situações em que estas variações ocorrem.

A Tabela 3.8 apresenta o *template* pelo qual foi definida a linha de processo. A Tabela 3.9 apresenta as informações básicas da LPS, bem como sua arquitetura, a qual foi projetada por meio da notação gráfica proposta por BARRETO [48]. Esta arquitetura denota a interconexão entre cada componente, além do tipo de conexão entre os mesmos, bem como a obrigatoriedade de cada componente e de cada conexão.

Tabela 3.9: Informações básicas da Linha de Processo para Soluções Orientadas a Serviço

Linha de Processo para Soluções Orientadas a Serviço

Descrição:

Esta linha de processos tem por propósito oferecer uma forma estruturada de construção de soluções orientadas a serviço, que enderece desafios inerentes a tal paradigma e forneça soluções aderentes aos princípios de SOA, flexíveis e de ágil resposta a requisitos inconstantes, por meio de práticas do XP.

Definido por:

UNIRIO

Características Atendidas:

- *Iteration Planning*
- Testes de aceitação
- TDD
- Programação em Pares
- Integração Contínua
- *Iteration Demonstration*
- Reuso
- Contrato
- Baixo Acoplamento

- Abstração
- Autonomia
- Ausência de estado
- Descoberta
- Composabilidade
- Equipes distribuídas
- Modificação de contrato
- Testes distribuídos
- Distribuição de serviços

Arquitetura da Linha de Processos:

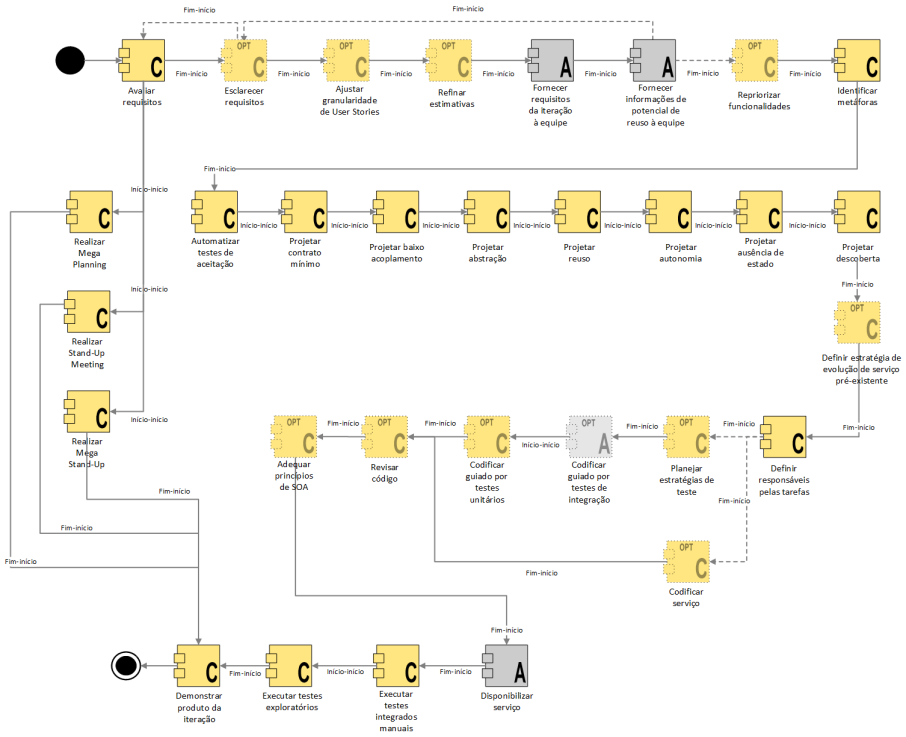


Tabela 3.8: *Template* utilizado para definição da LPS

<Nome da Linha de Processo de Software>
Descrição:
<Descrição da LPS>
Definido por:
<Organização que define a LPS>
Características Atendidas:
<Características atendidas pela LPS>
Arquitetura da Linha de Processos:
<Ilustração da LPS>

A fim de facilitar o entendimento da LPS em um nível mais amplo, os 41 (quarenta e um) componentes são representados de forma resumida na Tabela 3.10. Esta tabela apresenta os componentes definidos para a linha de processo, bem como suas especializações.

Tabela 3.10: Exemplo de componente concreto sem arquitetura interna

Componentes	**Variantes**
UNR.AVRE.CONC.0001 Avaliar requisitos	

UNR.ESRE.CONC.0002 Esclarecer requisitos	
UNR.REGR.CONC.0003 Ajustar granularidade das *User Stories*	
UNR.REES.CONC.0004 Refinar estimativas	
UNR.FORE.ABST.0005 Fornecer requisitos da iteração à equipe	UNR.APRE.CONC.0006 Apresentar requisitos da iteração à equipe
	UNR.DIRE.CONC.0007 Disponibilizar requisitos da iteração à equipe sem apresentação
UNR.FOIN.ABST.0008 Fornecer informações de potencial de reuso à equipe	UNR.APIN.CONC.0009 Apresentar informações de potencial de reuso à equipe
	UNR.DIRE.CONC.0010 Disponibilizar informações de potencial de reuso sem apresentação
UNR.ESRE.CONC.0011 Esclarecer requisitos	
UNR.REFU.CONC.0012 Repriorizar funcionalidades	
UNR.IDME.CONC.0013 Identificar metáforas	
UNR.AUTE.CONC.0014	

Automatizar testes de aceitação	
UNR.PRCO.CONC.0015 Projetar contrato mínimo	
UNR.PRBA.CONC.0016 Projetar baixo acoplamento	
UNR.PRAB.CONC.0017 Projetar abstração	
UNR.PRRE.CONC.0018 Projetar reuso	
UNR.PRAU.CONC.0019 Projetar autonomia	
UNR.PRAE.CONC.0020 Projetar ausência de estado	
UNR.PRDE.CONC.0021 Projetar descoberta	
UNR.DEES.CONC.0022 Definir estratégia de evolução de serviço pré-existente	
UNR.DERE.CONC.0023 Definir responsáveis pelas tarefas	
UNR.PLET.CONC.0024 Planejar estratégias de teste	
	UNR.COMA.CONC.0026 Codificar mocks de API para teste de

	integração entre serviços
UNR.COTI.ABST.0025 Codificar guiado por testes de integração	UNR.COMS.CONC.0027 Codificar mocks de serviços para teste de integração entre serviços
	UNR.COTR.CONC.0028 Codificar testes de integração entre serviços por meio de invocações reais
UNR.COTU.CONC.0029 Codificar guiado por testes unitários	
UNR.COSE.CONC.0030 Codificar serviço	
UNR.RECO.CONC.0031 Revisar código	
UNR.ADPS.CONC.0032 Adequar princípios de SOA	
UNR.DISE.CONC.0033 Disponibilizar serviço	
UNR.DIAU.CONC.0034 Disponibilizar serviço por meio automático	
UNR.DIMA.CONC.0035 Disponibilizar serviço manualmente	
UNR.EXTI.CONC.0036 Executar testes integrados manuais	

UNR.EXTE.CONC.0037 Executar testes exploratórios	
UNR.DEPI.CONC.0038 Demonstrar produto da iteração	
UNR.REMP.CONC.0039 Realizar mega planning	
UNR.RESM.CONC.0040 Realizar Stand-Up Meeting	
UNR.REMS.CONC.0041 Realizar Mega Stand-Up	

3.6Avaliação da linha de processo

A última etapa da estratégia *top-down* [48] diz respeito à avaliação da LPS, seus componentes e atividades, sob as óticas da sua adequação às necessidades identificadas e da aplicação correta da abordagem de definição.

BARRETO [48] prevê dois tipos de avaliação. A primeira se dedica a avaliar a correção na aplicação da abordagem de definição de processos para reutilização e a forma dos componentes, ao passo que a segunda objetiva avaliar o conteúdo dos componentes. A avaliação desta LPS se dedicou a ambos os tipos, por meio da técnica de **Revisão por Pares**, conforme proposto por BARRETO [48]. Para apoiar esta revisão, os revisores receberam o formulário de avaliação ilustrado na Figura 3.1, o qual se baseou no formulário utilizado por NUNES [23].

Foram convidadas quatro pessoas para a revisão da proposta, dentre as quais uma não pôde participar. Os demais revisores apresentam o seguinte perfil:

- Um profissional da IBM, com conhecimentos tanto em SOA quanto em métodos ágeis. Sua participação em projetos SOA se deu atuando em diversos papéis, como

Revisão por Pares da Linha de Processo

Instruções

1. Leia as definições dos componentes da linha de processo, analisando se as definições nela presentes contribuem para apoiar a construção de soluções orientadas a serviço por meio de práticas ágeis. Exemplos de critérios para avaliação dos componentes:
   a. *O componente está definido com clareza?*
   b. *O componente está aderente aos princípios ágeis?*
   c. *O componente está aderente aos princípios de SOA?*
   d. *A LPS favorece a geração de soluções orientadas a serviço de qualidade que sejam flexíveis e respondam com agilidade a novas prioridades de negócio?*
2. Durante a leitura, identifique pontos do conteúdo para os quais você deseja registrar um comentário.
3. Utilize a o formulário abaixo para registrar seus comentários.
4. Ao concluir sua revisão, envie o presente documento para felipe.carvalho@uniriotec.br.

Legenda para a coluna "Categoria" do formulário abaixo:
- **TA** *(Técnico Alto): significa que foi encontrado um problema em um item que compromete a utilização da LPS por equipes em projetos reais.*
- **TB** *(Técnico Baixo): significa que foi encontrado um problema em um item que seria conveniente alterar;*
- **E** *(Editorial): significa que foi encontrado um erro de português*
- **G** *(Geral): significa que o comentário é geral.*

Avaliador:	

ID do componente	Categoria (TA, TB, E, G)	Item dentro do componente	Comentário com a justificativa	Novo texto proposto ou observação

Figura 3.1: Formulário para Revisão por Pares

desenvolvedor, analista de requisitos e projetista por mais de cinco anos. Segundo sua própria avaliação, seu nível de conhecimento em SOA é classificado como médio, tendo participado de 2 a 5 projetos durante este período e trabalhando como professor de pós graduação em SOA. Por outro lado, esta mesma pessoa tem também mais de 5 anos de experiência com métodos ágeis, atuando como desenvolvedor, projetista, analista de requisitos e gerente de projetos, além de atuar como instrutor em cursos de métodos ágeis para funcionários da IBM. Segundo sua própria avaliação, esta pessoa é especialista em métodos ágeis.

- Um profissional da WDev, com menos de um ano de experiência em SOA, atuando no papel de analista de requisitos em 2 a 5 projetos. Este mesmo profissional tem de 2 a 5 anos de experiência com métodos ágeis, tendo participado em mais de 5 projetos, com atribuições diversas como Scrum Master, analista de requisitos e gerente de projetos.

- Uma pesquisadora da NP2Tec com pouca experiência tanto em SOA quanto métodos ágeis, tendo participado de um projeto SOA com duração entre 1 e 2 anos, e sem participação em projetos ágeis. No entanto, esta pesquisadora apresenta experiência na definição de componentes reutilizáveis de processos de *software*, tendo definido uma componentização de processos de *software*.

O pesquisador convidado com experiência em SOA e na definição de processos e componentes reutilizáveis fez a revisão da forma. Sua revisão teve por objetivo verificar

se os conceitos de reutilização de processos foram respeitados na definição e atendimento das características, além da criação de componentes, definição de seu sequenciamento e suas variantes.

Para a revisão do conteúdo, todos os convidados avaliaram a LPS proposta do ponto de vista de adequação ao seu propósito e de favorecer o fornecimento de soluções orientadas a serviço com qualidade, flexíveis e que respondam com agilidade a novas prioridades de negócio por meio de práticas do XP.

As respostas destes especialistas identificaram não conformidades relativas aos aspectos listados abaixo. Estes comentários foram utilizados na melhoria da LPS apresentada na dissertação.

Nesta avaliação, foram identificadas não conformidades relacionadas a:

- Necessidade de definição do conceito de "Pronto". Este é um importante conceito do Scrum, tendo sido definido na seção 3.1 e citado no componente UNR.DEPI.CONC.0038.
- Necessidade de esclarecimento de que, apesar da LPS se basear no XP, diversas práticas do Scrum são também utilizadas. Isto foi incorporado à seção 3.1.
- Necessidade de esclarecimento acerca do impacto trazido pela demora no esclarecimento de requisitos sobre o andamento da *Sprint*. Este esclarecimento foi acrescentado à atividade "Avaliar tempo necessário para esclarecer requisitos".
- Necessidade de acrescentar que o PO tem participação importante na identificação de metáforas. O componente UNR.IDME.CONC.0013 foi atualizado para refletir esta necessidade.
- Necessidade de esclarecer o impacto de uma atualização de serviço sobre sua documentação. A atividade "Notificar consumidores sobre nova versão de contrato" foi modificada para incorporar este item.
- Necessidade de verificar o encadeamento entre artefatos e critérios de entrada e saída entre os componentes. Com isso, as respectivas seções de diversos componentes foram corrigidas para utilizar artefatos e critérios com os mesmos nomes, conforme seu encadeamento.
- Necessidade de ajustar a granularidade das atividades, o que acarretou a eliminação da atividade "Corrigir critérios não atendidos", que constava da arquitetura interna do componente "UNR.ESRE.CONC.0002".

3.7Considerações finais

Este capítulo apresentou as etapas desempenhadas conforme a estratégia *top-down* proposta por BARRETO [48], a fim de definir a Linha de Processo de *Software* para Soluções Orientadas a Serviço, segundo os requisitos definidos para os propósitos desta dissertação.

Foram apresentados, por conseguinte, a definição das características a serem atendidas, bem como a definição, caracterização e estruturação dos componentes que endereçam estas características. Por fim, foram também apresentados a proposta de avaliação da LPS, bem como os resultados desta avaliação por meio da técnica de Revisão por Pares. Esta avaliação teve por objetivo aferir a qualidade e aderência dos componentes definidos quanto aos requisitos propostos e a técnica de reutilização de processos utilizada.

4. Conclusão

4.1Considerações finais

Num mundo em mudança constante e de competição globalizada, os modelos de negócio das organizações vem evoluindo e consolidando um modelo econômico baseado em serviços. Neste modelo, cada organização se estrutura em torno de suas competências centrais e estabelece parcerias para atender demandas complexas, de forma a aumentar suas vantagens competitivas.

No entanto, responder de forma rápida a requisitos instáveis e modelos de negócios mutáveis não é tarefa trivial. É necessário ter conhecimento profundo dos modelos de negócio e flexibilidade para responder com velocidade à necessidades em contínua modificação. Em resposta à necessidade de maior flexibilidade e menor tempo de resposta à mudanças, surgiram o conceito de SOA e métodos leves denominados “ágeis”.

Em paralelo, na Petrobras, restrições financeiras e necessidades de governança aumentaram o interesse por conceitos como reuso de soluções, desacoplamento, flexibilidade e agilidade na produção de novos produtos a partir de soluções pré-existentes. Ao mesmo tempo, mudanças culturais e uma busca por maior qualidade levaram a conclusão sobre a necessidade de se ter entregas parciais na entrega de sistemas desenvolvidos externa ou internamente, e maior integração com os clientes. Assim, os conceitos de SOA e métodos ágeis são fundamentais no momento atual da TIC da Petrobras, de forma a: (i) melhorar a comunicação entre os diversos interessados em um produto; (ii) melhorar a qualidade deste produto, (iii) diminuir o prazo de entrega de um produto por meio do reuso de soluções pré-existentes e (iv) diminuir o número de sistemas a serem monitorados e mantidos ativos.

No entanto, apesar de ambos terem surgido com o objetivo de responder a necessidades semelhantes, não há consenso na literatura sobre o uso de métodos ágeis para entregar

soluções orientadas a serviço. Autores divergem sobre a validade desta combinação e apontam dificuldades inerentes desenvolvimento de serviços. A maioria dos autores que oferecem uma visão sobre como se combinar estas abordagens, o faz por meio de diretrizes, mapeamentos, *insights*, entre outras formatos mais informais. Há poucos trabalhos que ofereçam uma forma estruturada de trabalho que uma equipe possa seguir para entregar soluções aderentes aos preceitos de orientação a serviço e métodos ágeis. Além disso, verificou-se que estes autores referenciavam com mais frequência atividades referentes ao planejamento e controle de projetos ágeis, do que atividades relativas a efetiva construção de software. Há poucas orientações sobre como endereçar dificuldades inerentes ao desenvolvimento de soluções orientadas a serviço, bem como boas práticas para times distribuídos. Por fim, estes autores não levaram em consideração o contexto de aplicação de tal método, quando existente, a despeito da importância da adequação de processos de *software* às necessidades das organizações.

Assim, este trabalho procurou definir uma linha de processo de *software*, com o objetivo de oferecer uma forma estruturada de trabalho através da qual sejam construídas soluções orientadas a serviço por meio de práticas ágeis do XP, com flexibilidade e agilidade suficiente para responder com velocidade à necessidades em constante evolução das organizações.

4.2Contribuições

A principal contribuição deste trabalho é uma forma estruturada de trabalho, a qual pode ser seguida por equipes trabalhando no desenvolvimento de soluções orientadas a serviço e customizada conforme a realidade de suas organizações e projetos. Esta forma é implementada através de uma linha de processo de *software*, construída sobre os princípios de SOA, as características de XP e lacunas identificadas na literatura no que se refere a métodos que combinem o uso de métodos ágeis na entrega de soluções orientadas a serviço. Além disso, esta LPS, por meio de seus componentes, endereça também boas práticas referentes a características de soluções orientadas a serviço, além de preocupações inerentes a times distribuídos, configuração comum de equipes de desenvolvimento SOA.

Outras contribuições foram geradas ao longo da pesquisa que deu origem a esta dissertação.

O artigo CARVALHO & AZEVEDO [17] traz duas contribuições. A primeira é a classificação dos métodos ágeis catalogados por STRODE [32] segundo uma taxonomia própria, baseada nas características destes métodos. Além desta, este trabalho também

apresenta um mapeamento entre as práticas documentadas por cada método ágil *versus* e as práticas adotadas por cada método. Desta forma, é possível verificar quais práticas são compartilhadas por quais métodos. Este mapeamento permitiu concluir que, entre os métodos ágeis catalogados por STRODE [32], o XP é aquele que apresenta maior abrangência no uso de práticas ágeis, utilizando aproximadamente 45% de todas as práticas catalogadas.

Por fim, o artigo CARVALHO, AZEVEDO & SANTOS [25] apresenta um estudo de mapeamento sistemático, apresentando os trabalhos disponíveis na literatura que tratam da combinação de SOA e XP, as lacunas encontradas na literatura e um método que preenche estas lacunas com o objetivo de oferecer uma forma estruturada de se construir soluções orientadas a serviço por meio de práticas do XP.

4.3Limitações

Apesar das contribuições citadas, é possível observar algumas limitações aos resultados deste trabalho. A primeira delas diz respeito a avaliação da proposta. Não foi possível avaliar esta proposta por meio do seu uso em um projeto real, nem coletar o *feedback* de pessoas utilizando uma instância da LPS proposta. Além disso, a dificuldade para contactar profissionais especialistas em definição de linhas de processos de *software*, levou à participação de apenas dois especialistas na revisão da LPS proposta. Em complemento, não foi possível encontrar especialistas com conhecimento conjunto nos três assuntos desta dissertação, a saber, definição de linhas de processos de *software*, SOA e métodos ágeis (mais especificamente, XP). Desta forma, a avaliação da LPS foi feita por pessoas com vieses específicos, ao invés de um viés mais generalista.

No que se refere a ameaças, há aquela relativa à garantia de que todos os componentes necessários ao domínio foram corretamente identificados e definidos. Esta ameaça é mitigada por meio de diretrizes, boas práticas e sugestões obtidas na literatura, além da própria experiência prática dos autores nos assuntos desta dissertação.

4.3.1Trabalhos futuros

Possibilidades de trabalhos futuros incluem:

- Aplicação prática da LPS em contextos reais da indústria;
- Refinamento e expansão dos componentes para melhor atendimento aos princípios

de SOA e XP, e aprimoramento da qualidade e adequação dos componentes;

- Investigação de outros pontos de variabilidade, a fim de atender contextos que não foram previstos no escopo desta dissertação.

Referências Bibliográficas

5. Referências bibliográficas

[1] KOHLBORN, T., KORTHAUS, A., CHAN, T., ROSEMANN, M., "Identification and analysis of business and software services - a consolidated approach". *IEEE Transactions On Services Computing*, v. 2, n. 1, pp. 50-64, 2009.

[2] LANKHORST, M., *Agile Service Development*. 1 ed. Berlin, Springer, 2012.

[3] JOSUTTIS, N., *SOA in Practice*. 1 ed. Sebastopol, O'Reilly, 2007.

[4] ERL, T., *SOA: Principles of Service Design*. 1 ed. Boston, Pearson, 2007.

[5] CHATERJEE, S., WEBBER, J., *Developing Enterprise Web Services: An Architect's Guide*. 1 ed. New Jersey, Prentice Hall, 2003.

[6] GU, Q., LAGO, P., "A Stakeholder-Driven Service Life Cycle Model for SOA". In: *Proceedings of 2nd International Workshop On Service oriented software engineering: in conjunction with the 6th ESEC/FSE joint meeting*, pp. 1-7, Cavat, Set, 2007.

[7] PRESSMAN, R.; MAXIM, B., *Enterprise Integration Patterns: designing, building and deploying messaging solutions*. 16 ed. Boston, Addison-Wesley, 2012.

[8] GU, Q., LAGO, P., "On Service-Oriented Architectural Concerns and Viewpoints". In: *Software Architecture, 2009 & European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on*, pp. 289-292, Set, Cambridge, 2009

[9] AgileAlliance, 2001, *Manifesto for agile software development*. In: http://www.agilemanifesto.org. Acessado em Abr/2012.

[10] KROGDAHL, P., LUEF, G., STEINDL, C., "Service-oriented agility: an initial analysis for the use of agile methods for SOA development". In: *Proceedings of*

the 2005 IEEE International Conference on Services Computing (SCC'05), pp. 93-100, Jul, Orlando, 2005.

[11] MARKS, E.A., BELL, M., *Service-oriented architecture. A planning and implementation guide for business and technology*. 1 ed. New Jersey, John Wiley & Sons, 2006.

[12] FOWLER, M., 2008, *Can SOA Be Done With An Agile Approach?*. In: http://martinfowler.com/bliki/EvolutionarySOA.html. Acessado em Ago/2014.

[13] VIVACQUA, F. R., *Fábricas de Software e a academia: análise da formação acadêmica em informática no município do Rio de Janeiro*. Dissertação de M.Sc., FGV, Rio de Janeiro, 2009.

[14] TAN, A., ANG, C. H., LEE, E. W., HAINES, M.,2005, *Web Service Implementation Methodology: Case Example using Extreme Programming*. In: https://www.oasis-open.org/committees/download.php/14027/fwsi-im-1.0-XPExample-doc-wd-04.doc. Acessado em Jun/2014.

[15] KARSTEN, P., CANNIZZO, F., "The creation of a distributed agile team". In: *Agile Processes in Software Engineering and Extreme Programming*, pp.235-239, Como, Jun, 2007

[16] BECK, K., *Extreme Programming Explained*. 1 ed. New York, Addison-Wesley, 1999.

[17] CARVALHO, F.; AZEVEDO, L. G. "Service Agile Development Using XP". In: *Service Oriented System Engineering (SOSE), 2013 IEEE 7th International Symposium on*, pp. 254-259, Redwood City, Mar, 2013.

[18] LEE, E. W., TAN, P. S., CHENG, Y., "Web Service Implementation Methodology", Organization for the Advancement of Structured Information Standards (OASIS), 2005.

[19] KIRCHER, M., JAIN, P., CORSARO, A., LEVINE, D., "Distributed extreme programming". In: *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering*, pp. 66-71, Villasimius, Maio, 2001.

[20] MARANZATO, R., NEUBERT, M., HERCULANO, P., "Scaling Scrum Step by Step: 'The Mega Framework"'. In: *Agile Conference (AGILE), 2012*, pp. 79-85, Dallas, Ago, 2012.

[21] ABRANTES, J.; TRAVASSOS, G. [2011]. "Common Agile Practices in Software Processes". In: *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on*, pp. 355-358, Banff, Set, 2011.

[22] LINDVALL, M.; BASILI, V.; BOEHM, B.; COSTA, P.; DANGLE, K.; SHULL, F.; TESORIERO, R.; WILLIAMS, L.; ZELKOWITZ, M. [2002]. "Empirical Findings in Agile Methods". In: *Extreme Programming and Agile Methods—XP/Agile Universe 2002*, pp. 197-207, Chicago, Ago, 2002.

[23] NUNES, E., *Definição de Processos de Aquisição de Software para Reutilização*. Dissertação de M.Sc., PESC/COPPE, Rio de Janeiro, 2011.

[24] NASSIR, M.; SAHIBUDDIN, S. [2011]. "Critical Success Factors for Software Projects: A Comparative Study", *Scientific Research and Essays*, v. 6, n. 10, pp. 2174-2186, 2011.

[25] CARVALHO, F.; AZEVEDO, L. G.; SANTOS, G. "A Method for Service Agile Construction". *Simpósio Brasileiro de Sistemas de Informação*, 124387, Londrina, Maio, 2014.

[26] ARMBRUST, O.; KATAHIRA, M.; MIYAMAMOTO, Y.; MÜNCH, J.; NAKAO, H.; OCAMPO, A. "Scoping Software Process Lines", *Software Process: Improvement and Practice*, v. 14, n. 3, pp. 181-197, 2009.

[27] NUNES, E.; ROCHA, A. R.; SANTOS, G. [2012]. "Definição de Processos de Aquisição de Software com Uso de Abordagem baseada em Reutilização". In: *Simpósio Brasileiro de Qualidade de Software*, pp. 1-15, Fortaleza, Jun, 2012.

[28] HOHPE, G.; WOOLF, B., *Enterprise Integration Patterns: designing, building and deploying messaging solutions*. 16 ed. Boston, Addison-Wesley, 2012.

[29] KRAFZIG, D., BANKE, K., SLAMA, D., *Enterprise SOA: Service-Oriented Architecture Best Practices*. 1 ed. New Jersey, Prentice Hall, 2004.

[30] World Wide Web Consortium, "Simple Object Access Protocol (SOAP)". Disponível em `http://www.w3.org/TR/soap/`. Acessado em 11/Ago/2014.

[31] BANAVAR, G., CHANDRA, T., STROM, R., STURMAN, D., "A Case for Message Oriented Middleware". In: *Proceedings of the 13th International Symposium on Distributed Computing*, pp. 1-18, London, 1999.

[32] STRODE, D. E., "Agile methods: a comparative analysis". In: *Proc. 19th Annual Conference of the National Advisory Committee on Computing Qualifications (NACCQ 2006), Wellington, New Zealand*, Wellington, Jul, 2006.

[33] COCKBURN, A. "Humans and Technology", Cutter IT Journal, 2001. Disponível em `http://alistair.cockburn.us/Crystal+light+methods`. Acessado em 11/Ago/2014.

[34] ABRAHAMSSON, P., WARSTA, J., SIPONEN, M., RONKAINEN, J., "New Directions on Agile Methods: A Comparative Analysis". In: *Software Engineering, 2003. Proceedings. 25th International Conference on*, pp. 244-254, Portland, Maio, 2003.

[35] ABRAHAMSSON, P., SALO, O., RONKAINEN, J.,2002, *Agile Software development methods review and analysis, 2002*. In: VTT Publications, v. 478.

[36] VOIGT, B., GLINZ, M., SEYBOLD, C., *Dynamic System Development Method*, Department of Information Technology, University of Zurich, 2004

[37] COCKBURN, A., *Surviving object-oriented projects: a manager's guide*. 1 ed. New York, Addison Wesley, 1998.

[38] KRUCHTEN, P., "A Rational Development Process", *Crosstalk*, v. 9, n. 7, pp. 11-16, 1996.

[39] XProgramming, 1999, *An Agile Software Development Resource*. In: http://xprogramming.com/. Acessado em Abr/2012.

[40] HIGHSMITH, J.A., *Adaptative Software Development: A Collaborative Aproach to Managing Complex Systems*. 1 ed.New York, Dorset House, 2000.

[41] BASKERVILLE, R., RAMESH, B., "Is Internet-Speed Software Development Different?", *IEEE Software*, v. 20, n. 6, pp. 70-77, 2003.

[42] POPPENDIECK, M., "Lean Software Development". In: *Companion to the proceedings of the 29th International Conference on Software Engineering*, pp. 165-166, Minneapolis, Maio, 2007

[43] POPPENDIECK, M., POPPENDIECK, T., *Lean Software Development: An Agile Toolkit*. 1 ed. New York, Addison-Wesley, 2003.

[44] MELO, C., SANTOS, V., CORBUCCI, H., KATAYAMA, E., GOLDMAN, A., KON, F., *Métodos Ágeis no Brasil: Estado da Prática em Times e Organizações*. Relatório Técnico RT-MAC-2012-03, USP, Maio, 2012.

[45] PARNAS, DL. "On the Design and Development of Program Families", *Software Engineering, IEEE Transactions on*, n. 1, pp. 1-9, 1976.

[46] TEIXEIRA, E.N., *Odysseyprocess-Fex: Uma Abordagem para Modelagem de Variabilidades de Linha de Processos de Software*. Dissertação de M.Sc., Programa de Pós-graduação em Engenharia de Sistemas e Computação - PESC/COPPE, UFRJ, Rio de Janeiro, 2011.

[47] BARRETO, A. S., MURTA, L. G. P., ROCHA, A. R. "Componentizando Processos Legados de Software Visando a Reutilização de Processos". In: *Simpósio Brasileiro de Qualidade de Software*, pp. 189-203, Ouro Preto, 2009.

[48] BARRETO, A.S., *Uma Abordagem para Definição de Processos Baseada em Reutilização Visando à Alta Maturidade em Processos*. Tese de D.Sc., Programa de Pós-graduação em Engenharia de Sistemas e Computação - PESC/COPPE, UFRJ, Rio de Janeiro, 2011.

[49] ESTUBLIER, J., DAMI, S., "About reuse in multi-paradigm process modelling approach". In: *Proceedings of the International Software Process Workshop*, pp. 63-65, Dijon, Jun, 1996.

[50] COSTA, A., SALES, E., REIS, C.A.L., *et al.*, "Apoio a Reutilização de Processos de Software através de Templates e Versões". In: *VI Simpósio Brasileiro de Qualidade de Software*, pp. 47-61, Porto de Galinhas, Jun, 2007.

[51] XU, P., "Knowledge Support in Software Process Tailoring". In: *38th Hawaii International Conference on System Sciences, pp. 1-9*, Honolulu, Jan, 2005.

[52] CARDOSO, F. S., *Definição de processos reutilizáveis para projetos com aquisição*. Dissertação de M.Sc., Programa de Pós-graduação em Engenharia de Sistemas e Computação - PESC/COPPE, UFRJ, Rio de Janeiro, 2012.

[53] ARSANJANI, A., GHOSH, S., ALLAM, A., ABDOLLAH, T., GANAPATHY, S., HOLLEY, K., 2008, "SOMA: A method for developing service-oriented solutions", *IBM systems Journal*, v. 47, n. 3, pp. 377-396, 2008.

[54] DIIRR, T., AZEVEDO, L. G., SANTORO, F., BAIÃO, F., FARIA, F., "Practical Approach for Service Design and Implementation". In: *Proceedings of IAIS Information Systems*, v. 1, pp. 197-204, Berlin, Mar, 2012.

[55] IVANYUKOVICH, A., GANGADHARAN, G. R., D'ANDREA, V., MARCHESE, M. "Towards a service-oriented development methodology", *Journal of Integrated Design and Process Science*, v. 9, n. 3, pp. 53-62, 2005.

[56] WAKE, W. C., *Extreme programming explored*. 1 ed. New York, Addison-Wesley, 2002.

[57] SCRUM.ORG, `https://www.scrum.org/Scrum-Guide`.

[58] SCHWABER, K., BEEDLE, M., *Agile Software Development with Scrum*. 1 ed, New York, Prentice Hall, 2001.

[59] COHN, M., "Differences Between Scrum and Extreme Programming", `http://www.mountaingoatsoftware.com/blog/differences-between-scrum-and-extreme-programming`.

[60] JEFFRIES, R., ANDERSON, A., HENDRICKSON, C., *Extreme Programming Installed*. 1 ed. New York, Addison-Wesley, 2000.

[61] COHN, M., *User Stories Applied: For Agile Software Development*. 1 ed. New York, Addison-Wesley, 2004.

[62] BECK, K., FOWLER, M., *Planning eXtreme Programming Explained*. 1 ed. New York, Addison-Wesley, 2000.

[63] KITCHENHAM, B., "Procedures for performing systematic reviews", *Keele, UK, Keele University*, v. 33, p. 2004, 2004.

[64] BASILI, V. R., ROMBACH, H. D., "The TAME project: towards improvement-oriented software environments", *Software Engineering, IEEE Transactions on*, v. 14, n. 6, pp. 758–773, 1988.

[65] CUSUMANO, M., "Software Development on Internet Time", *IEEE Computer*, v. 32, n. 10, pp. 60-69, 1999.

[66] ARMBRUST, O., KATAHIRA, M., MIYAMAMOTO, Y., MÜNCH, J., NAKAO, H., OCAMPO, A., "Scoping Software Process Lines", *Software Process: Improvement and Practice*, v. 14, n. 14, pp. 181-197, 2009.

[67] DORAN, G. T. "There's a S.M.A.R.T. way to write management's goals and objectives", *Management Review*, v. 70, n. 11, pp. 35–36, 1981.

[68] KEELING, M., VELICHANSKY, M., "Making Metaphors that Matter", In: Agile Conference, Salt Lake City, Ago, 2011.

[69] AZEVEDO, L. G., BAIÃO, F. A., SANTORO, F., SOUZA, J., REVOREDO, K., PEREIRA, V., HERLAIN, I., "Identificação de Serviços a partir da Modelagem de

Processos de Negócio". *Simpósio Brasileiro de Sistemas de Informação*, Brasília, Maio, 2009.

[70] NARAYANAN, V., 2010, *Modern SOA methodology and SOA adoption using agile practices*. In: http://www.servicetechmag.com/I42/0810-2. Acessado em Jul/2014.

[71] LOPES, S., MELE, M., *Uma Abordagem Prática para Desenvolvimento de Web Services com Contract-First*. Trabalho de Conclusão de Curso, Bacharelado em Sistemas de Informação, Universidade Federal do Estado do Rio de Janeiro, 2012.

[72] GILPIN, M., *From The Field: The First Annual Canonical Model Management Forum*, 2010. In: `http://blogs.forrester.com/print/mike_gilpin/10-03-15-field_first_annual_canonical_model_management_forum`. Acessado em Jul/2014.

[73] OLIVEIRA, J. A., RAMOS, F. G., DIAS JUNIOR, J. J. L., "Reusabilidade em SOA: Um Mapeamento Sistemático da Literatura", *Simpósio Brasileiro de Sistemas de Informação*, Londrina, Maio, 2014.

[74] CURBERA, F., DUFTLER, M., KHALAF, R., NAGY, W., MUKHI, N., WEEWARANA, S., "Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI", *IEEE Internet computing* v. 6, n. 2, pp. 86-93, 2002.

[75] RIBAROV, L., MANOVA, I., ILIEVA, S., 2007, "Testing in a service-oriented world". In: *Proceedings of the International Conference on Information Technologies (InfoTech-2007)*, v. 2, Set, Varna, 2007.

6. Apêndice A - Estudo de Mapeamento Sistemático

Este apêndice apresenta um estudo de mapeamento sistemático, complementado por *snowballing*, cujo objetivo foi investigar métodos pré-existentes na literatura que lançassem mão de de práticas de XP para a construção de soluções orientadas a serviço aderentes aos princípios de SOA.

6.1Protocolo de pesquisa

O protocolo de pesquisa utilizado neste estudo de mapeamento sistemático se baseia nos padrões estabelecidos por KITCHENHAM [63] e se encontra detalhado nas seções a seguir. O orientador da dissertação, especialista em SOA, avaliou o protocolo e os resultados.

6.1.1Contexto

Há diversas propostas na literatura de uso de práticas do eXtreme Programming (XP) em combinação aos princípios de Service-Oriented Architecture (SOA). Este estudo tem por objetivo identificar que práticas, técnicas e princípios de ambos tem sido usados em combinação, se estas estão documentadas na forma de métodos concretos e repetíveis, e como estes tem sido avaliados.

6.1.2Objetivo - paradigma GQM (Goal - Question - Metric) [64]

- **analisar** publicações científicas sobre o uso combinado de práticas de XP e princípios de SOA
- **com o propósito de** identificar métodos concretos e repetíveis que implementem esta combinação, além de formas de avaliação

- **com relação a** métodos ágeis de desenvolvimento SOA
- **do ponto de vista de** pesquisador
- **nos contextos** acadêmico e industrial

6.1.3Critérios de inclusão

- **QI1**: o artigo trata do uso combinado de SOA com XP?
- **QS1**: o artigo trata do uso de práticas ágeis comuns ao XP em combinação a SOA?

6.1.4Questões de pesquisa

- **QP**: que métodos foram propostos para combinar princípios de SOA e práticas de XP?
- **QS1**: como estes métodos foram avaliados?
- **QS2**: o método é concreto e repetível?
- **QS3**: o método trata a fase de construção de serviços?

6.1.5Escopo

A pesquisa foi guiada pelos **critérios de inclusão**, utilizando como fontes:

1. as seguintes ferramenta de busca:
   (a) Scopus
   (b) DBLP Complete Search
   (c) Compendex
   (d) ACM Digital Library
   (e) IEEExplore
2. os anais das seguintes conferências, em suas edições de 2005 a 2012:
   (a) SBSI
   (b) ICEIS
   (c) AMCIS

(d) CAiSE

(e) ICIS

(f) ECIS

(g) SOSE

(h) SBCARS

(i) ESELAW

(j) XP Conference

Os artigos analisados foram publicados entre 2000 e 2013, e englobaram somente a área de ciência da computação.

6.1.6Idioma

Inglês

6.1.7Métodos de busca de publicações

As ferramentas de busca foram acessadas via *Web*, usando a seguinte expressão de busca, em formato independente de ferramenta: *((soa or service oriented) AND (xp or "extreme programming"))*

Os anais das conferências foram analisados manualmente, uma vez que muitas não dispunham de motores de busca. Com acessos aos anais, foram feitas buscas pelas expressões "xp", "extreme programming", "soa", "service oriented" sobre os títulos e *abstracts* dos trabalhos.

6.1.8Procedimentos de seleção e critérios

A seleção de artigos foi feita em 3 etapas:

1. Seleção preliminar de artigos: tratou da execução dos métodos de busca de publicações;
2. Seleção de abstracts relevantes: os abstracts foram analisados para identificar se os trabalhos tratavam da combinação de SOA e XP;
3. Análise dos trabalhos relevantes: uma vez eliminados os trabalhos com abstracts que não se relacionam contexto do trabalho, os restantes foram analisados para responder às perguntas de pesquisa.

Após a terceira etapa, foram aplicados procedimentos de *snowballing* [**?**], a fim de complementar os resultados obtidos com outras publicações relevantes referenciadas por algum artigo selecionadona segunda ou terceira etapas. A análise das referências foi feita de forma iterativa, até que não houvesse novas referências relevantes [**?**]. Os seguintes critérios de inclusão foram aplicados:

4. Ser referenciado por alguma publicação escolhida ao fim da segunda ou terceira etapas de seleção;
5. Atender aos critérios de seleção da segunda e terceira etapas;

6.1.9Procedimentos para extração dos dados

Os seguintes dados foram extraídos dos trabalhos relevantes:

- Título;
- Autores;
- Ano de publicação;
- Resumo do método proposto;
- Forma de avaliação do método;
- Fases tratadas pelo método;

6.1.10Procedimentos para análise

Os dados foram analisados de forma quantitativa, a partir do número de publicações selecionadas para compor o estudo. Estes números embasam a análise qualitativa posterior, que usa os resultados da busca para responder as questões de pesquisa.

6.1.11Testes do protocolo

A primeira bateria de execução utilizou a seguinte expressão de busca: *((soa or service oriented) AND (xp or "extreme programming"))*. Esta execução retornou 7 resultados no Scopus, 32 no IEEExplore, 21 no Compendex, 0 no DBLP e 29 no ACM DL. Consideramos a ausência de resultados no DBLP bastante incomum, sendo feitas outras tentativas com combinações diversas da expressão de busca, mas nenhum resultado foi encontrado.

Devido ao baixo número de resultados no Scopus, modificamos a expressão de busca para *((soa or service oriented) AND (xp or "extreme programming"))*. Esta mudança trouxe 12 novos resultados no Scopus, 3 no IEEExplore, 23 no Compendex, 0 no DBLP e 3 no ACM DL.

Estes números foram considerados melhores, mas tentamos uma nova combinação, acrescentando as siglas SOSE (que significa *Service-Oriented Software Engineering*) e SOC (*Service-Oriented Computing*): *((soa or service oriented or SOSE or SOC) AND (xp or "extreme programming"))*.

Esta modificação trouxe prejuízos aos resultados da pesquisa, devido ao grande número de falsos positivos relativos a sigla "SOC", associada também a expressão *System-on-Chip*, utilizada no contexto de desenvolvimento de sistemas embarcados. Assim, consideramos a seguinte expressão ótima para os propósitos da pesquisa: *((soa or service oriented) AND (xp or "extreme programming"))*.

Após identificar os artigos retornados pelos motores de busca, os anais das conferências foram analisados manualmente, tomando por base a mesma expressão de busca, conforme descrito anteriormente. Ao final, foram identificados 153 artigos para análise.

6.2Execução da pesquisa

Uma vez estabelecido o protocolo, a pesquisa foi executada, estando a execução do protocolo datada de Jan/2013.

Conforme os procedimentos de seleção e critérios de inclusão descritos à seção 6.1.8, a primeira etapa de seleção consistiu da execução da expressão de busca sobre máquinas de busca e anais de conferências detalhados na seção 6.1.5. Nesta etapa foram retornados 153 artigos.

A segunda etapa consistiu da leitura dos *abstracts* dos artigos retornados, a fim de eliminar trabalhos que não se relacionassem com os temas de SOA e XP. Ao final da análise dos *abstracts*, 30 artigos foram selecionados para a fase seguinte.

A terceira etapa se dedicou a analisar o texto completo dos trabalhos restantes para identificar os trabalhos relevantes para responder às questões de pesquisa. Ao final desta etapa, sete artigos foram selecionados como relevantes.

Além disso, algumas publicações foram analisadas por meio de *snowballing*, levando à inclusão de cinco outros artigos à segunda fase, onde três destes passaram para a terceira

fase.

Os 158 artigos selecionados são apresentados na Tabela 6.1. Note-se que a sigla **2E** identifica os artigos selecionados para a segunda fase, enquanto a sigla **3E** identifica os aprovados à terceira fase. A sigla **SA** indica artigos que foram eliminados por não estarem disponíveis para *download*. Por sua vez, a Tabela 6.2 apresenta os nove trabalhos classificados como relevantes após as três etapas.

6.2.1Análise do resultado da pesquisa

A partir das informações obtidas por meio da execução do protocolo de pesquisa, verificou-se que somente o trabalho da OASIS [14] respondia a alguma pergunta de pesquisa. Mais especificamente, este método responde à todas as questões de pesquisa, exceto à questão **QS1**.

Verificou-se que os demais autores com frequência não definiam fases, papéis, responsabilidade ou entregáveis. Muitos apresentavam ideias, diretrizes gerais ou mapeamentos entre conceitos de SOA e XP, mas não ofereciam métodos concretos. A maioria dos autores também não endereçava a fase de construção de serviços.

6.2.2Resultados da pesquisa

Esta seção apresenta o resultado final da seleção dos trabalhos a partir da execução do protocolo de pesquisa em Janeiro de 2013.

6.2.2.1Publicações retornadas

A Tabela 6.1 apresenta os 158 trabalhos analisados, onde os primeiros 153 foram retornados pela execução da expressão de busca e os últimos 5 foram acrescentados por *snowballing*.

Os nove artigos selecionados ao final das etapas de seleção são destacados na Tabela 6.2.

Tabela 6.1: Listagem de publicações retornadas

Autor(es)	**Título**	**Ano**	**2E**	**3E**

Lin, L., Yang, W., Lin, J.	A layer-based method for rapid software development	2012	S	N
Mathrani, A., Mathrani, S.	Test strategies in distributed software development environments	2013	N	N
Xu, J., Townend, P., Webster, D.	Interface Refactoring in Performance-Constrained Web Services	2012	S	N
Zimmermann, O., Miksovic, C., Kusster, J.M.,	Reference architecture, metamodel, and modeling principles for architectural knowledge management in information technology services	2012	N	N
Crowder, R., Fowler, D., Reul, Q., Sleeman, D., Shadbolt, N., Wills, G.	An information system to support the engineering designer	2012	N	N
Janes, A.A., Succi, G.	The dark side of agile software development	2012	N	N
Bagchi, T.P.	A note on risks in software development with imperfect testing	2011	S	N
Miksovic, C., Zimmermann, O.	Architecturally Significant Requirements, Reference Architecture, and Metamodel for Knowledge Management in Information Technology Services	2011	N	N

	Technical program / abstract	2011	SA	N
Lv, J., Li, X.Y., Han, Y.F., Liang, Y.J., Zhang, Z.J.	A scheme of embedded video surveillance system software architecture design	2011	N	N
Fung, K. H., Low, G. C.	Quality factors for dynamic evolution in composition-based distributed applications	2011	N	N
Brand, E.A., Honig, W.L., Wojtowicz, M.	Intelligent systems development in a non engineering curriculum	2011	N	N
Franky, M.C.	Agile management and development of software projects based on collaborative environments	2011	N	N
Kim, W.Y., Park, S.G.	The 4-tier design pattern for the development of an android application	2011	N	N
Karekar, C., Tarrell, A., Fruhling, A.	Agile Development at ABC – What Went Wrong?	2011	S	SA
Read, A., Callens, A., Nguyen, C., de Vreede, G. J.	Generating User Stories in Groups with Prompts	2011	N	N

Schrodl, H., Wind, S.	Adoption of Scrum for Software Development Projects: An Exploratory Case Study from the ICT Industry	2011	N	N
Bergkvist, L., Johansson, B.	Management of Information Systems Outsourcing: Challenges and Lessons Learned	2011	N	N
Roy, S., Debnath, M.K.	Designing SOA based e-governance system using eXtreme Programming methodology for developing countries	2010	N	N
Jaatun, M.G., Jensen, J., Sasson, R.	The Road to Hell is Paved with Good Intentions: A Story of (In)secure Software Development	2010	S	N
Al-Dabass, D., Guha, R.	Impact of Web 2.0 and Cloud Computing Platform on Software Engineering	2010	S	S
Graf, F., Maiden, N., Seyff, N.	Using Mobile RE Tools to Give End-Users Their Own Voice	2010	N	N
Parveen, T., Tilley, S.	When to Migrate Software Testing to the Cloud?	2010	N	N
Ganis, M., Maximilien, E. M., Rivera, T.	A brief report on working smarter with Agile software development	2010	S	N

Moawad, R., Zaki, K.M.	A hybrid disciplined Agile software process model	2010	S	S
Kuhn, E., Mordinyi, R., Schatten, A.	Towards an Architectural Framework for Agile Software Development	2010	N	N
Nylund, H., Andersson, P.H.	Simulation of service-oriented and distributed manufacturing systems	2010	N	N
Andrikopoulos, V., Bucchiarone, A., Di Nitto, E., Kazhamiakin, R., Lane, S., Mazza, V., Richardson, I.	Service engineering	2010	SA	N
Hasegawa, M., Bandara, U., Inoue, M., Morikawa, H.	A network oriented simple device and its application in a service mobility environment	2004	N	N
Onions, P., Patel, C.	Enterprise SoBA: Large-scale implementation of Acceptance Test Driven Story Cards	2009	S	S
Lubke, D., Salnikow, A., Singer, L.	Calculating BPEL test coverage through instrumentation	2009	N	N
Banjanin, M., Guikers, R., Kayser, A., Ketter, W.	Introducing an Agile Method for Enterprise Mash-Up Component Development	2009	S	N

Barzilay, O., Hazzan, O., Yehudai, A.	A Multidimensional Software Engineering Course	2009	N	N
Dalle, O., Ribault, J., Himmelspach, J.	Design considerations for M&S software	2009	N	N
SIGSOFT Software Engineering Notes	ACM SIGSOFT Software Engineering Notes: Volume 34 Issue 2	2009	N	N
SIGSOFT Software Engineering Notes	ACM SIGSOFT Software Engineering Notes: Volume 34 Issue 3	2009	N	N
Vasilecas, O., Smaizys, A.	Business rule model integration into the model of transformation driven software development	2009	N	N
Dunbar, C., Fruhling, A., McDonald, P.	A Case Study: Introducing eXtreme Programming in a US Government System Development Project	2008	S	N
Gschwind, T., Leymann, F., Zdun, U., Zimmermann, O.	Combining Pattern Languages and Reusable Architectural Decision Models into a Comprehensive and Comprehensible Design Method	2008	N	N
	The Big Bang: 25 Years of Software History	2008	N	N
	Table of contents	2008	SA	N

Cannizzo, F., Marcionetti, G., Moser, P.	The Toolbox of a Successful Software Craftsman	2008	S	N
Bahsoon, R., Emmerich, W.	An Economics-Driven Approach for Valuing Scalability in Distributed Architectures	2008	N	N
Qumer, A., Henderson-Sellers, B.	A framework to support the evaluation, adoption and improvement of agile methods in practice	2008	S	N
Dexter, H., Petch, J., Powley, D.	Establishing a development process for composite applications in the work-based learning and competency management domain	2008	N	N
Nikolov, R., Ilieva, S.	A model for strengthening the software engineering research capacity	2008	N	N
Hacker, W.	Intersection of software methodologies and ITIL V3	2008	N	N
Nagappan, N., Maximilien, E.M., Bhat, T., Williams, L.	Realizing quality improvement through test driven development: results and experiences of four industrial teams	2008	N	N
Aguiar, A., Yoder, J.	Proceedings of the 15th Conference on Pattern Languages of Programs	2008	N	N

	Queue: Volume 6 Issue 5	2008	N	N
Karsten, P., Cannizzo, F.	The creation of a distributed agile team	2007	S	S
An, J.B.C., Chung, S., Davalos, S.	Service-Oriented Software Reengineering: SoSR	2007	N	N
Abbas, N., Davis, H.C., Gilbert, L., Howard, Y., Millard, D.E., Walters, R.J., Wills, G.B.	The Service Responsibility and Interaction Design Method: Using an Agile Approach for Web Service Design	2007	N	N
Haines, M.N.	The Impact of Service-Oriented Application Development on Software Development Methodology	2007	S	N
Vossen, G., Hagemann, S.	Unleashing Web 2.0: From Concepts to Creativity	2007	N	N
Qumer, A., Henderson-Sellers, B.	An agile toolkit to support agent-oriented and service-oriented computing mechanisms	2007	S	N
Wills, G.B., Abbas, N., Chandrasekharan, R., Crowder, R.M., Gilbert, L., Howard, Y., Millard, D.E., Wong, S.C., Walters, R.J.	An agile hypertext design methodology	2007	N	N

Liu, X., Li, C., Ma, Z., Wang, J., Sun, W.	A super data-sharing model in common platform of geographic information	2011	N	N
Lee, E.W., Chan, L.P., Lee, S.P.	Web Services Implementation Methodology for SOA Application	2006	S	S
Saurer, G., Schatten, A., Schiefer, J.	Testing complex business process solutions	2006	N	N
Shan, T.C., Hua, W.W.	Solution Architecting Mechanism	2006	N	N
Demirkan, H., Goul, M., Keith, M., Mitchell, M.C., Nichols, J.	Contextualizing Knowledge Management Readiness to Support Change Management Strategies	2006	S	N
	Editorial archives	2006	SA	N
Betz, C.T.	Architecture and Patterns for IT Service Management, Resource Planning, and Governance: Making Shoes for the Cobbler's Children: Making Shoes for the Cobbler's Children, 2nd edition	2006	N	N
Guttman, M., Parodi, J.	Real-Life MDA: Solving Business Problems with Model Driven Architecture	2006	N	N

Mayer, P., Lubke, D.	Towards a BPEL unit testing framework	2006	N	N
Lubke, D., Schneider, K.	Leveraging feedback on processes in SOA projects	2006	S	N
Aiello, G., Alessi, M., Cossentino, M., Urso, A., Vella, G.	RTDWD: real-time distributed wideband-delphi for user stories estimation	2006	S	N
Krogdahl, P., Luef, G., Steindl, C.	Service-oriented agility: an initial analysis for the use of agile methods for SOA development	2005	S	S
Jones, S.	Toward an acceptable definition of service [service-oriented architecture]	2005	S	N
Steindl, C.	From agile software development to agile businesses	2005	S	N
Akinaga, T., Kakimoto, T., Matsumoto, K., Monden, A., Ohsugi, N., Tsunoda, M.	Recommendation of software technologies based on collaborative filtering	2005	N	N
Douce, C., Livingstone, D., Orwell, J.	Automatic test-based assessment of programming: A review	2005	N	N
Ivanyukovich, A., Gangadharan, G.R., D'Andrea, V., Marchese, M.	Towards a service-oriented development methodology	2005	S	S

Morkel Theunissen, W. H., Boake, A., Kourie, D.G.	In search of the sweet spot: agile open collaborative corporate software development	2005	N	N
Lin, J., Lin, C., Yang, W.	An integrated method for rapid software development and effective process management	2013	S	N
Vinoski, S.	Do you know where your architecture is?	2003	S	N
	Enabling Technologies for the New Knowledge Society: ITI 3rd International Conference on Information & Communications Technology (ICICT 2005)	2005	N	N
Rasmusson, J.	Introducing XP into greenfield projects: Lessons learned	2003	S	N
Li, Q., Pan, X., Hou, C., Jin, Y., Dai, H., Wang, H., Zhao, X., Liu, X.	Exploring the dependence of bulk properties on surface chemistries and microstructures of commercially composite RO membranes by novel characterization approaches	2012	N	N

Liu, M., Yao, G., Cheng, Q., Ma, M., Yu, S., Gao, C.	Acid stable thin-film composite membrane for nanofiltration prepared from naphthalene-6-trisulfonylchloride (NTSC) and piperazine (PIP)	2012	N	N
Hubert, J., Dufour, T., Vandencasteele, N., Desbief, S., Lazzaroni, R., Reniers, F.	Etching processes of polytetrafluoroethylene surfaces exposed to He and He-O2 atmospheric post-discharges	2012	N	N
Jiang, D.	Study on development and application of platform with test questions library and paper generation based on SOA	2011	N	N
Keeling, M., Velichansky, M.	Making metaphors that matter	2011	S	N
Tort, A., Oliva, A., Sancho, M.R.	An approach to test-driven development of conceptual schemas	2011	N	N
Tran, T.K., Bricaud, Q., Oafrain, M., Blanchard, P., Roncali, J., Lenfant, S., Godey, S., Vuillaume, D., Rondeau, D.	Thiolate chemistry: A powerful and versatile synthetic tool for immobilization/functionalization of oligothiophenes on a gold surface	2011	N	N

Pang, X.R., Xiangbin K., Fei X., Jiandong H.	Inhibiting effect of ciprofloxacin, norfloxacin and ofloxacin on corrosion of mild steel in hydrochloric acid	2010	N	N
Wu, D., Liu, X., Yu, S., Liu, M., Gao, C.	Modification of aromatic polyamide thin-film composite reverse osmosis membranes by surface coating of thermo-responsive copolymers P(NIPAM-co-Am). I: Preparation and characterization	2010	N	N
Gebremariam, B., Duguet, T., Bogner, S.K.	Symbolic integration of a product of two spherical Bessel functions with an additional exponential and polynomial factor	2010	N	N
Gebremariam, B., Bogner, S.K., Duguet, T.	Symbolic computation of the Hartree-Fock energy from a chiral EFT three-nucleon interaction at N2LO	2010	N	N
Yu, S., Liu, M., Liu, X., Gao, C.	Performance enhancement in interfacially synthesized thin-film composite polyamide-urethane reverse osmosis membrane for seawater desalination	2009	N	N

Yu, S., Liu, M., Li, Z. Zhou, Y., Gao, C.	Aromatic-cycloaliphatic polyamide thin-film composite membrane with improved chlorine resistance prepared from m-phenylenediamine-4-methyl and cyclohexane-5-tricarbonyl chloride	2009	N	N
Liu, M., Wu, D., Yu, S., Gao, C.	Influence of the polyacyl chloride structure on the reverse osmosis performance, surface properties and chlorine stability of the thin-film composite polyamide membranes	2009	N	N
Colombo, P., Del Bianco, V., Lavazza, L.	Fine-grained integrated management of software configurations and traceability relations	2008	N	N
Meihong, L., Sanchuan, Y., Yong, Z., Congjie, G.	Study on the thin-film composite nanofiltration membrane for the removal of sulfate from concentrated salt aqueous: Preparation and performance	2008	N	N

Liu, M., Yu, S., Tao, J., Gao, C.	Preparation, structure characteristics and separation properties of thin-film composite polyamide-urethane seawater reverse osmosis membrane	2008	N	N
Demoisson, F., Raes, M., Terryn, H., Guillot, J., Migeon, H., Noel Reniers, F.	Characterization of gold nanoclusters deposited on HOPG by atmospheric plasma treatment	2008	N	N
Mariga, J.	Utilizing virtual software to provide hands-on experience with systems and applications software for is students	2007	N	N
Chow, T., Tsai, W.T., Balasooriya, J., Bai, X.	Ontology-based information sharing in Service-Oriented Database Systems	2009	N	N
	Proceedings - Testing: Academic and Industrial Conference - Practice and Research Techniques, TAIC PART 2006	2006	N	N
Yong, Z., Sanchuan, Y., Meihong, L., Congjie, G.	Polyamide thin film composite membrane prepared from m-phenylenediamine and m-phenylenediamine-5-sulfonic acid	2006	N	N

Liu, L.F., Yu, S.C., Zhou, Y. Gao, C.J.	Study on a novel polyamide-urea reverse osmosis composite membrane (ICIC-MPD). I. Preparation and characterization of ICIC-MPD membrane	2006	N	N
	Proceedings of the 2006 International Workshop on Interdisciplinary Software Engineering Research, WISER '06, Co-located with the 28th International Conference on Software Engineering, ICSE 2006	2006	N	N
	Proceedings of the 28th Annual International Computer Software and Applications Conference, Workshops and Fast Abstracts-COMPSAC 2004	2004	N	N
Labadie, J.W.	MODSIM: Decision support system for integrated river basin management	2006	N	N
Brox, M., Sanchez-Solano, S.	Development of IP modules of fuzzy controllers for the design of embedded systems on FPGAS	2006	N	N

Zhou, Y., Yu, S., Liu, M., Gao, C.,	Preparation and characterization of polyamide-urethane thin-film composite membranes	2005	N	N
LaFon, R.	ViewSonic VP231 wb	2005	N	N
	Proceedings of the Second IASTED International Conference on Communications, Internet, and Information Technology	2003	N	N
Succi, G., Stefanovic, M., Pedrycz, W.	Quantitative assessment of extreme programming practices	2001	N	N
	Enabling Technologies for the New Knowledge Society: ITI 3rd International Conference on Information &amp, Communications Technology (ICICT 2005)	2005	N	N
Gorton, I.	XML does real programmers a service	2008	N	N
Kim, J., Hoi Kim, H., Soo Sim, E., Deok Kim, K., Ho Kwon, O., Kee Oh, K.R.	Filter-free wavelength conversion using mach-zehnder interferometer with integrated multimode interference semiconductor optical amplifiers	2004	N	N

	Proceedings of the 28th Annual International Computer Software and Applications Conference, Workshops and Fast Abstracts-COMPSAC 2004	2004	N	N
Shibata, Y., Kikuchi, N., Tohmori, Y.	Photonic Integrated Devices for High Speed Signal Processing and Switching	2003	N	N
Nouveau, C., Djouadi, M.A., Deces-Petit, C., Beer, P., Lambertin, M.	Influence of CrxNy coatings deposited by magnetron sputtering on tool service life in wood processing	2001	N	N
	23rd Edition of the International Semiconductor Conference (CAS 2000)	2000	N	N
Cao, Y., Yu, L., Liu, W.	Study of the tribological behavior of sulfurized fatty acids as additives in rapeseed oil	2000	N	N
Dubinsky, Y., Hazzan, O., Talby, D., Keren, A.	System Analysis and Design in a Large-Scale Software Project: The Case of Transition to Agile Development	2006	S	N

Jiang, D.	Study on development and application of platform with test questions library and paper generation based on SOA	2011	N	N
Pang J. F., Tong D., Li H., He L., Cheng X.	PKUsim-86: A cycle level full system simulator for x86 processor and AMBA-based SoC	2011	N	N
Huang K., Lu J., Pang J., Zheng Y., Li H., Tong D., Cheng X.	FPGA prototyping of an AMBA-based windows-compatible SoC	2010	N	N
Jayadevappa S., Shankar R.	The changing ways of computer science & engineering education: A suitable pedagogy to adapt better	2009	N	N
Zhang Z.J., Jiang X.L., Sun C.X., Hu J.L., Yuan J.H.	DC pollution flashover circuit model of polluted insulator string	2009	N	N
Chuluunbaatar O., Gusev A.A., Gerdt V.P., Rostovtsev V.A., Vinitsky S.I., Abrashkevich A.G., Kaschiev M.S., Serov V.V.	POTHMF: A program for computing potential curves and matrix elements of the coupled adiabatic radial equations for a hydrogen-like atom in a homogeneous magnetic field	2008	N	N
Bacila, A., Decoopman, X., Mesmacque, G., Voda, M., Serban, V.A.	Study of underload effects on the delay induced by an overload in fatigue crack propagation	2007	N	N

Ragab, R., Malash, N., Gawad, G.A., Arslan, A., Ghaibeh, A.	A holistic generic integrated approach for irrigation, crop and field management: 2. The SALTMED model validation using field data of five growing seasons from Egypt and Syria	2005	N	N
Araujo G., Barros E., Melcher E., Azevedo R., Da Silva K.R.G., Prado B., De Lima M.E.	A SystemC-only design methodology and the CINE-IP multimedia platform	2005	N	N
Ukolov Y.A., Chekanov N.A., Gusev A.A., Rostovtsev V.A., Vinitsky S.I., Uwano Y.	A REDUCE program for the normalization of polynomial Hamiltonians	2005	N	N
Youn, S.W., Kang, C.G.	Maskless pattern fabrication on Pyrex 7740 glass surface by using nano-scratch with HF wet etching	2005	N	N
Erdelyi, T.	Markov- and Bernstein-Type Inequalities for Müntz Polynomials and Exponential Sums in Lp	2000	N	N
Sitaraman, S.	On a Fermat-type Diophantine equation	2000	N	N

Scheithauer, M., Bosch, E., Schubert, U.A., Knozinger, H., Cheung, T.-K., Jentoft, F.C., Gates, B.C., Tesche, B.	Spectroscopic and microscopic characterization of iron- and/or manganese-promoted sulfated zirconia	1998	N	N
Yinong, C., Tsai, W.T.	Service-orientation in computing curriculum	2011	N	N
Yinong, C., Zhihui, D., Garcia-Acosta, M.	Robot as a Service in Cloud Computing	2010	N	N
Tsai, W.T.	Service-oriented system engineering: a new paradigm	2005	S	N
Tomayko, J.E.	Teaching eXtreme programming remotely	2005	N	N
Huang, K., Lu, J., Pang, J., Zheng, Y., Li, H., Tong, D., Cheng, X.	FPGA prototyping of an AMBA-based windows-compatible SoC	2010	N	N
Xia, M., Yu, M., Lin, Q., Qi, Z., Guan, H.	Enhanced privilege separation for commodity software on virtualized platform	2010	N	N
Irrera, I., Duroes, J. ,Vieira, M., Madeira, H.	Towards identifying the best variables for failure prediction using injection of realistic software faults	2010	N	N
Viegas, V., Girao, P., Silva Pereira, J.M.	Open controller for distributed instrumentation systems	2009	N	N

Niu, X., Shu, X.,	Characterization of the creep constitutive behavior of SnAgCu solder in flip chip joints from the indentation creep testing	2010	N	N
Araujo, G., Barros, E., Melcher, E., Azevedo, R., Da Silva, K., Prado, B., De Lima, M.E.	A SystemC-only design methodology and the CINE-IP multimedia platform	2013	N	N
Nakamura, N., Takahama, S., Barolli, L., Ma, J., Sugita, K.	A multiplatform P2P system: Its implementation and applications	2005	N	N
Ukolov, Y., Chekanov, N.A., Gusev, A.A., Rostovtsev, V.A., Vinitsky, S.I., Uwano, Y.	A REDUCE program for the normalization of polynomial Hamiltonians	2005	N	N
Canals, J.A., MartInez, M.A., Ballester, F.J., Mora, A.	New FPSoC-based architecture for efficient FSBM motion estimation processing in video standards	2007	N	N
Jayaram, V., Hegde, M.S., Reddy, K.P.J.	Synthesis of carbon nitride by interaction of carbon C60 with strong shock heated nitrogen gas in free piston driven shock tube	2009	N	N
Canals, J.A., Martinez, M.A., Ballester, F.J.	FPSoC-based architecture for efficient FSBM motion estimation processing	2006	N	N

Kuehn, E., Sieck, J.	Design and implementation of location and situation based services for a pervasive mobile adventure game	2009	N	N
Jayadevappa, S., Shankar, R.	The changing ways of computer science &amp, engineering education: A suitable pedagogy to adapt better	2009	N	N
Choudhary, M.K., Kasprzak, C., Larson, R.H., Venuturumilli, R.	ASHRAE standard 90.1 metal building U-factors - Part 1: Mathematical modeling and validation by calibrated hot box measurements	2010	N	N
Tan, S.Y., Wu, M.Y., Chen, H.H., Hsia, Y.L.	Characterizing the interfacial properties of HfO2/Si and HfSiO/Si gate stacks	2009	N	N
Tian, X.S., Wang, Q., Sun, J.F., Fan, Z.G.	The calculation of vanadium dioxide thin films optical properties at 10.6 micron	2012	N	N
Tan, A., Ang, C. H., Lee, E. W., Haines, M	OASIS FWSI IMSC Web Services Implementation Methodology - Case Example using Extreme Programming	2005	S	S

Dubinsky, Y., Hazzan, O., Keren, A.	Introducing Extreme Programming into a Software Project at the Israeli Air Force	2005	S	N
Kicher, M., Jain, P., Corsaro, A., Levine, D.	Distributed eXtreme Programming	2001	S	S
Maurer, F.	Supporting Distributed Extreme Programming	2002	S	N
Patel, C., Ramachandran, M.	Acceptance Test Driven Story Card Development for XP	2008	S	N

Tabela 6.2: Listagem de publicações relevantes

Autor(es)	**Título**	**Ano**
Al-Dabass, D., Guha, R.	Impact of Web 2.0 and Cloud Computing Platform on Software Engineering	2010
Moawad, R., Zaki, K.M.	A hybrid disciplined Agile software process model	2010
Onions, P., Patel, C.	Enterprise SoBA: Large-scale implementation of Acceptance Test Driven Story Cards	2009
Karsten, P., Cannizzo, F.	The creation of a distributed agile team	2007
Lee, E.W., Chan, L.P., Lee, S.P.	Web Services Implementation Methodology for SOA Application	2006
Krogdahl, P., Luef, G., Steindl, C.	Service-oriented agility: an initial analysis for the use of agile methods for SOA development	2005

Ivanyukovich, A., Gangadharan, G.R., D'Andrea, V., Marchese, M.	Towards a service-oriented development methodology	2005
Tan, A., Ang, C. H., Lee, E. W., Haines, M	OASIS FWSI IMSC Web Services Implementation Methodology - Case Example using Extreme Programming	2005
Kicher, M., Jain, P., Corsaro, A., Levine, D.	Distributed eXtreme Programming	2001

6.2.2.2Informações extraídas das publicações selecionadas

Esta seção apresenta as informações extraídas dos nove artigos selecionados ao final do estudo de mapeamento sistemático.

Título	Impact of Web 2.0 and Cloud Computing Platform on Software Engineering
Autor(es)	Al-Dabass, D., Guha, R.
Ano	2010
Resumo da publicação	Fala sobre a inclusão do papel do *Cloud Provider* no ciclo de desenvolvimento, mas não relaciona as práticas de XP. Não detalha, por exemplo, se o *Cloud Provider* participa do *Planning Game*. O método proposto se intitula uma extensão do XP, mas não cita nenhuma de suas práticas (ex.: *Planning Game*).
Como é avaliado?	Não foi avaliado.
É concreto e repetível?	Não é um método concreto ou repetível.

Trata a fase de construção de serviços?	Não.

Título	A hybrid disciplined Agile software process model
Autor(es)	Moawad, R., Zaki, K. M.
Ano	2010
Resumo da publicação	A fase de construção é tratada, mas faz propostas controversas. Por exemplo, o item 3, na página 5, *Continuous testing cost*, propõe práticas que contradizem os princípios ágeis, entre outras controvérsias também relativas a fase de construção.
Como é avaliado?	Não foi avaliado.
É concreto e repetível?	Não é um método concreto ou repetível.
Trata a fase de construção de serviços?	Sim.

Título	Enterprise SoBA: Large-scale implementation of Acceptance Test Driven Story Cards
Autor(es)	Onions, P., Patel, C.
Ano	2009

Resumo da publicação	O artigo fala sobre o uso de critérios de aceitação com SOA, mas não entra em detalhes sobre a utilização da técnica.
Como é avaliado?	Não foi avaliado.
É concreto e repetível?	Não é um método concreto ou repetível.
Trata a fase de construção de serviços?	Não.

Título	The creation of a distributed agile team
Autor(es)	Karsten, P., Cannizzo, F.
Ano	2007
Resumo da publicação	O artigo discorre sobre o uso de times ágeis distribuídos na construção de uma API orientada a serviços. Fala sobre boas práticas relativas a organização de times distribuídos, disseminação de conhecimento, priorização de *Backlog*, uso de testes automatizados de aceitação, utilização de PO's e ferramental para o dia a dia de times distribuídos.
Como é avaliado?	Não foi avaliado.
É concreto e repetível?	Não é um método concreto ou repetível.

Trata a fase de construção de serviços?	Sim.

Título	Web Services Implementation Methodology for SOA Application
Autor(es)	Lee, E. W., Chan, L. P., Lee, S. P.
Ano	2006
Resumo da publicação	Define um conjunto de boas práticas a serem observadas durante a construção de serviços, levando em consideração complexidades inerentes a soluções orientadas a serviço, como, por exemplo, escolha do tipo de ligação, uso de tipos primitivos para favorecer a interoperabilidade, entre outros itens.
Como é avaliado?	Não foi avaliado.
É concreto e repetível?	Não é um método concreto ou repetível.
Trata a fase de construção de serviços?	Sim.

Título	Service-oriented agility: an initial analysis for the use of agile methods for SOA development
Autor(es)	Krogdahl, P., Luef, G., Steindl, C.
Ano	2005

Resumo da publicação	Os autores fazem uma contraposição entre os princípios de SOA e os princípios de Lean e Scrum, abordando priorização de funcionalidades, priorização de elementos de arquitetura, disseminação de informação em times distribuídos e refatoração de serviços num ambiente corporativo. No entanto, os pontos abordados são apenas ideias gerais, não estão ancorados em experiência prática.
Como é avaliado?	Não foi avaliado.
É concreto e repetível?	Não é um método concreto ou repetível.
Trata a fase de construção de serviços?	Não.

Título	Service-oriented agility: an initial analysis for the use of agile methods for SOA development
Autor(es)	Ivanyukovich, A., Gangadharan, G. R., D'Andrea, V., Marchese, M.
Ano	2005
Resumo da publicação	O artigo traça os primeiros passos em direção ao que parece ser um método que endereça SOA, XP e RUP. Os autores estabelecem paralelos entre conceitos ágeis e conceitos de SOA, além de delinear o impacto trazido por práticas do XP sobre o desenvolvimento de serviços.

Como é avaliado?	Não foi avaliado.
É concreto e repetível?	Não é um método concreto ou repetível.
Trata a fase de construção de serviços?	Não.

Título	OASIS FWSI IMSC Web Services Implementation Methodology - Case Example using Extreme Programming
Autor(es)	Tan, A., Ang, C. H., Lee, E. W., Haines, M
Ano	2005
Resumo da publicação	Evolução do trabalho de LEE *et al.* [18], este artigo trata da expansão do método proposto por LEE *et al.* [18] para utilizar práticas do XP no desenvolvimento de soluções orientadas a serviço. No entanto, a evolução proposta deixa de abordar diversas práticas importantes do XP, como codificação guiada por testes, uso de iterações e o uso de testes de aceitação automatizados.
Como é avaliado?	Não foi avaliado.
É concreto e repetível?	Sim.
Trata a fase de construção de serviços?	Sim.

Título	Distributed eXtreme Programming
Autor(es)	Kicher, M., Jain, P., Corsaro, A., Levine, D.
Ano	2001
Resumo da publicação	O autor explicita as práticas de XP mais diretamente impactadas por times distribuídos (*Planning Game*, *Pair Programming*, *On-site Customer*) e relata como as contornou em um projeto de curta duração.
Como é avaliado?	Não foi avaliado.
É concreto e repetível?	Não é um método concreto ou repetível.
Trata a fase de construção de serviços?	Sim.

6.3Considerações finais

Este apêndice apresentou um estudo sobre métodos que utilizam práticas de XP para a construção de soluções orientadas a serviço, por meio de estudo de mapeamento sistemático, complementado por *snowballing*.

O estudo buscou identificar propostas pré-existentes na literatura que endereçassem a construção de serviços aderentes aos princípios de SOA e que empregasse práticas do XP ao longo de seu ciclo de desenvolvimento. A execução deste estudo levou a conclusão de que a maioria dos autores apresenta ideias, mapeamentos de conceitos e diretrizes gerais de uso de práticas de XP num ciclo de desenvolvimento SOA, sem definir um método concreto e repetível.

Foi encontrado somente um artigo que propusesse um método concreto e repetível, que tratasse da fase de construção de serviços e endereçasse o uso de práticas do XP,

publicado por TAN *et al.* [14], no entanto, este método lança mão de poucas práticas do XP, deixando de considerar aspectos importantes e fundamentais, como, por exemplo, codificação guiada por testes, uso de iterações e o uso de testes de aceitação automatizados.

O protocolo foi revisado por um pesquisador com experiência na condução e revisão de estudos similares, no entanto, há ameaças a validade do estudo.

A primeira ameaça diz respeito à publicações que não estivessem disponíveis nas fontes de dados utilizadas e que não tenham sido capturadas pela aplicação de *snowballing*. Tais publicações podem ser relevantes ao estudo, afetar a generalização dos resultados e ameaçar a validade externa.

A composição da expressão de busca pode igualmente ameaçar a validade externa, uma vez que publicações relevantes podem ter sido excluídas dos resultados da busca por não apresentarem os termos selecionados. Esta ameaça é tratada pela execução do *snowballing*.

Por fim, uma ameaça a validade interna advém da análise qualitativa dos *abstracts*, executada na segunda etapa de seleção dos estudos. Uma análise equivocada do descrito nos *abstracts* pode ter levado a exclusão de trabalhos relevantes.

7. Apêndice B - Detalhamento da LPS

Este apêndice apresenta a Linha de Processo de Software para Soluções Orientadas a Serviço proposta nesta dissertação.

Ao longo da descrição dos componentes, haverá menção a itens identificados por meio de letras entre parênteses (ex.: **(a)**, **(b)**). Estes marcadores fazem menção a considerações sobre o uso daquele componente em um contexto real, descritas no Apêndice 8.

Linha de Processo para Soluções Orientadas a Serviço

Descrição:

Esta linha de processos tem por propósito oferecer uma forma estruturada de construção de soluções orientadas a serviço, que enderece desafios inerentes a tal paradigma e forneça soluções aderentes aos princípios de SOA, flexíveis e de ágil resposta a requisitos inconstantes, por meio de práticas do XP.

Definido por:

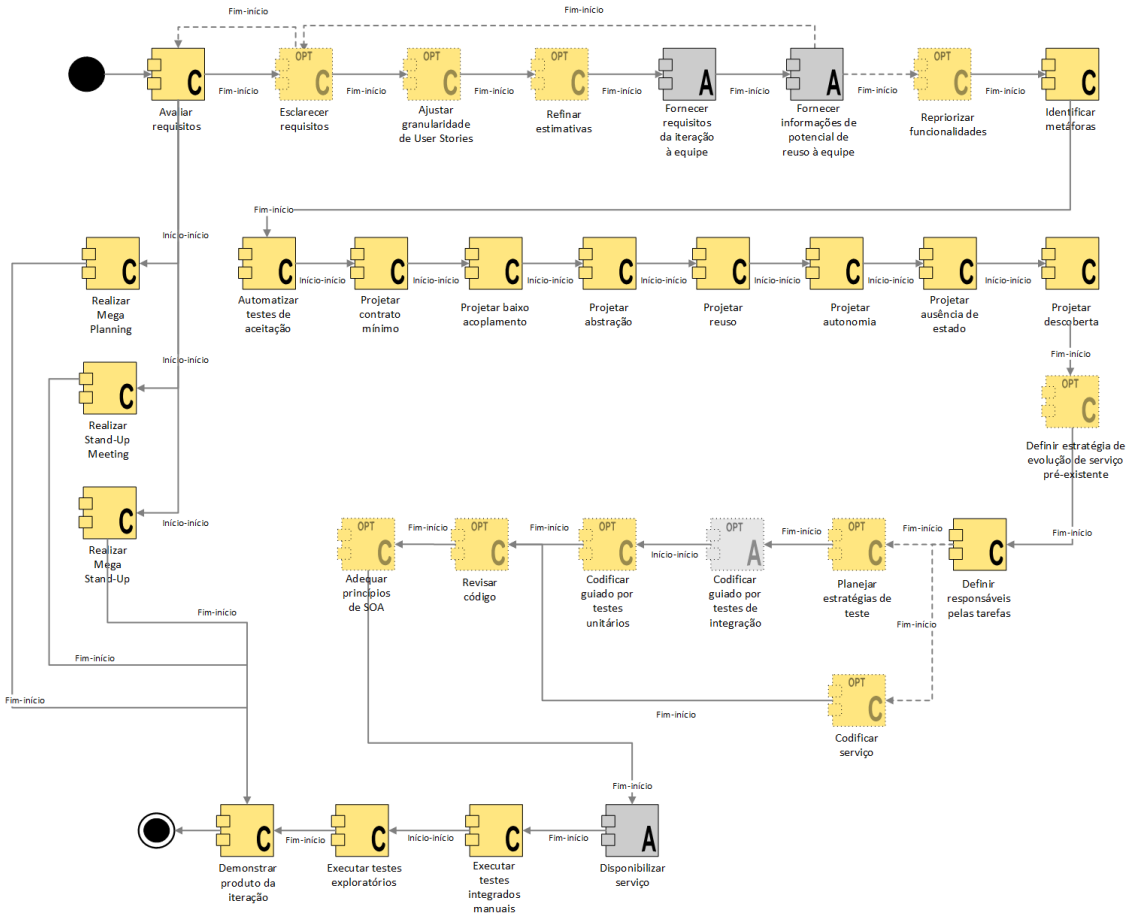
UNIRIO

Características Atendidas:

- *Iteration Planning*
- Testes de aceitação
- TDD

- Programação em Pares
- Integração Contínua
- *Iteration Demonstration*
- Reuso
- Contrato
- Baixo Acoplamento
- Abstração
- Autonomia
- Ausência de estado
- Descoberta
- Composabilidade
- Equipes distribuídas
- Modificação de contrato
- Testes distribuídos
- Distribuição de serviços

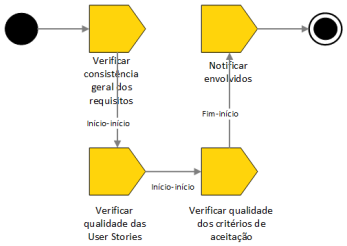
7.1Arquitetura da LPS



7.2Componentes e Atividades da LPS

Identificador	**UNR.AVRE.CONC.0001**
Nome:	Avaliar requisitos
Descrição:	Este componente tem por propósito entender e avaliar a validade e completude dos requisitos funcionais e não-funcionais da funcionalidade a ser desenvolvida. Pontos a serem esclarecidos devem ser registrados em local acordado com o PO **(a)**.

	A equipe fica responsável por destacar um ou mais membros para serem responsáveis pela avaliação dos requisitos da próxima iteração e estabelecer o prazo máximo **(b)** em que esta avaliação deve ser completada. Estes membros ficam responsáveis por decidir, dentro do prazo estabelecido, em que momento esta validação ocorrerá, e por obter os recursos necessários a sua execução (ex.: reserva de salas de reunião, ferramentas de áudio/videoconferência, *laptops* etc.) de acordo com a forma de trabalho da equipe (co-localizada ou distribuída). O trabalho de validação se divide em diversas atividades, conforme descrito na seção Arquitetura Interna a seguir.
Tipo de Componente:	Concreto
Obrigatório:	Sim
Participantes:	• PO
Responsáveis	• Equipe de desenvolvimento
Critérios de Entrada	• Levantamento de requisitos finalizado
Critérios de Saída	• Requisitos avaliados pela equipe
Artefatos Requeridos:	• *User Stories* • Critérios de aceitação definidos pelo PO
Artefatos Produzidos:	• Notificação da equipe de desenvolvimento ao PO quanto aos requisitos a serem esclarecidos; ou,

	• Notificação da equipe de desenvolvimento ao PO quanto aos requisitos que estão claros.
Características Atendidas:	• *Iteration Planning*
Variantes deste Componente:	Não há
Arquitetura Interna:	
Atividade	**Verificar consistência geral dos requisitos**
Descrição:	Nesta atividade, os membros da equipe devem verificar se: • os requisitos são coerentes com o entendimento inicial da equipe sobre a funcionalidade a ser desenvolvida; • há requisitos conflitantes; • os objetivos de negócio do cliente estão sendo alcançados; • há impacto sobre parceiros de negócio ou outras funcionalidades; • há dúvidas sobre o funcionamento esperado de alguma funcionalidade; Ao final desta atividade, os problemas e dúvidas encontrados devem ser registrados por meio de formato e ferramenta previamente acordados entre PO e equipe de desenvolvimento.
Participantes:	• Equipe de desenvolvimento

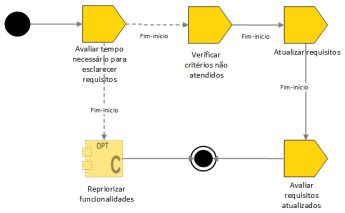
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• *User Stories* • Critérios de aceitação definidos pelo PO
Artefatos Produzidos:	• Registro dos problemas encontrados na verificação dos requisitos
Atividade	**Verificar qualidade das *User Stories***
Descrição:	Nesta atividade, deve-se verificar a qualidade das *User Stories*. Segundo COHN [61], a qualidade de *User Stories* pode ser medida segundo seis atributos, os quais são agrupados sob o acrônimo INVEST, definido abaixo: • "I" – *independent*: indica que as *User Stories* não apresentam dependências entre si; • "N" – *negotiable*: indica que uma *User Story* não funciona como um contrato, ou seja, indica que a *User Story* define detalhes em um nível de granularidade tal que permite sua mudança conforme o entendimento da mesma evolui; • "V" – *valuable*: indica que as *User Stories* agregam valor ao negócio do cliente; • "E" – *estimatable*: indica que a *User Story* tem um nível de detalhe e granularidade adequados, permitindo que sejam providas estimativas confiáveis a partir do que ela define; • "S" – *small*: indica que as *User Stories* têm granularidade e coesão adequadas; • "T" – *testable*: indica que as *User Stories* são passíveis de serem testadas;

	Ao final desta atividade, os participantes devem registrar, por meio de formato e ferramenta acordados, os atributos que não foram atendidos.
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• *User Stories*
Artefatos Produzidos:	• Registro dos atributos não atendidos na verificação das *User Stories*
Atividade	**Verificar qualidade dos critérios de aceitação**
Descrição:	Os critérios de aceitação representam os requisitos de qualidade do produto a ser construído. Assim, nesta atividade, a equipe deve verificar a qualidade dos critérios de aceitação. O acrônimo SMART define os seguintes atributos para avaliar a qualidade de critérios de aceitação. SMART foi proposto por DORAN [67] e é utilizado por COHN [61] e WAKE [56], e amplamente aceito na comunidade. • "S" – *specific*: indica que os critérios têm objetivos explícitos e bem definidos; • "M" – *measurable*: indica que são mensuráveis; • "A" – *achievable*: indica que, ao final da execução do critério, o usuário alcançará um resultado; • "R" – *relevant*: indica que os critérios são relevantes à *User Story*; • "T"- *time-boxed*: definem em que momento um resultado de negócio será alcançado.

	Ao final da atividade, as violações encontradas nos critérios de aceitação quanto ao modelo SMART devem ser registradas.
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Critérios de aceitação definidos pelo PO
Artefatos Produzidos:	• Registro dos atributos não atendidos na verificação dos critérios de aceitação
Atividade	**Notificar envolvidos**
Descrição:	Ao final da validação, os participantes são responsáveis por: • consolidar os problemas, violações e pontos a serem esclarecidos; • notificar PO e demais membros da equipe quanto aos pontos levantados nas atividades anteriores, respeitando os procedimentos previamente acordados para registro de dúvidas; ou, • notificar a todos caso não haja pontos a serem esclarecidos e os requisitos sejam considerados claros e maduros, e que o desenvolvimento pode ser iniciado imediatamente.
Participantes:	• PO
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Registro dos problemas encontrados na verificação dos requisitos • Registro dos atributos não atendidos na verificação das *User Stories*

	• Registro dos atributos não atendidos na verificação dos critérios de aceitação
Artefatos Produzidos:	• Notificação da equipe de desenvolvimento ao PO quanto aos pontos a serem esclarecidos; ou, • Notificação da equipe de desenvolvimento ao PO de que os requisitos estão claros.
Identificador	**UNR.ESRE.CONC.0002**
Nome:	Esclarecer requisitos
Descrição:	Este componente tem por propósito esclarecer os pontos levantados pela equipe de desenvolvimento, além de corrigir as violações aos modelos SMART e INVEST identificadas durante a execução do componente UNR.AVRE.CONC.0001. Se o PO optar pela execução deste componente, fica responsável por executar as atividades identificadas na seção Arquitetura Interna. Caso o PO opte pela não-execução deste componente, ele deve se responsabilizar, perante a equipe de desenvolvimento, por impactos decorrentes de esclarecimentos pendentes. Após o esclarecimento dos requisitos, a equipe de desenvolvimento e PO devem decidir em conjunto se os esclarecimentos gerados efetivamente eliminaram as dúvidas e violações, em cujo caso o fluxo de trabalho seguirá para o próximo componente; ou se uma nova rodada de esclarecimentos é necessária, sendo o componente UNR.AVRE.CONC.0001 re-executado. Tipicamente, um número elevado de esclarecimentos e/ou violações, ou pontos de esclarecimento extensos ou complexos indicam que a re-execução do componente UNR.AVRE.CONC.0001 pode ser necessária.

	O PO também tem a opção de não produzir esclarecimento ou correções imediatamente, repriorizando funcionalidades. Neste caso, o componente UNR.REFU.CONC.0012 será executado.
Tipo de Componente:	Concreto
Obrigatório:	Não
Participantes:	• Equipe de desenvolvimento
Responsáveis	• PO
Critérios de Entrada	• Avaliação de requisitos finalizada
Critérios de Saída	• Requisitos esclarecidos pelo PO
Artefatos Requeridos:	• *User Stories* • Critérios de aceitação • Notificação da equipe de desenvolvimento ao PO quanto aos pontos a serem esclarecidos
Artefatos Produzidos:	• *User Stories* e/ou critérios de aceitação atualizados para esclarecer os pontos solicitados; ou, • Notificação do PO à equipe de que se responsabiliza por impactos decorrentes de esclarecimentos pendentes, caso tenha optado pelos requisitos não serem validados; ou, • *Backlog* atualizado refletindo a mudança de priorização.
Características Atendidas:	• *Iteration Planning*

Variantes deste Componente:	Não há
Arquitetura Interna:	
Atividade	**Avaliar tempo necessário para esclarecer requisitos**
Descrição:	O PO deve avaliar a complexidade e quantidade de pontos a serem esclarecidos, em relação ao tempo que tem disponível para produzir esclarecimentos **(a)** e decidir se esclarece os pontos de dúvida imediatamente ou reprioriza funcionalidades. Note-se que, no caso da primeira *Sprint*, o início da mesma pode impactado, caso haja demora excessiva no esclarecimento dos requisitos.
Participantes:	• PO
Responsáveis	• PO
Artefatos Requeridos:	• Registro dos problemas encontrados na verificação dos requisitos
Artefatos Produzidos:	• Caso o PO não disponha de tempo suficiente para o esclarecimento dos requisitos, deve registrar que a iteração será re-priorizada.
Atividade	**Verificar critérios não atendidos**
Descrição:	O PO deve avaliar a consistência dos critérios que foram identificados como não atendidos, no que se refere às *User Stories* e critérios de aceitação, identificando e separando aqueles que devem ser acatados ou descartados.
Participantes:	• PO

Responsáveis	• PO
Artefatos Requeridos:	• Registros dos critérios que não foram atendidos
Artefatos Produzidos:	• Registros dos critérios que não foram atendidos como acatados ou descartados;
Atividade	**Atualizar requisitos**
Descrição:	O PO deve esclarecer as dúvidas e atualizar os requisitos (*User Stories* e critérios de aceitação) com os esclarecimentos para os pontos levantados.
Participantes:	• PO
Responsáveis	• PO
Artefatos Requeridos:	• Registro dos problemas encontrados na verificação dos requisitos
Artefatos Produzidos:	• *User Stories* e/ou critérios de aceitação atualizados para esclarecer os problemas encontrados na verificação dos requisitos
Atividade	**Avaliar requisitos atualizados**
Descrição:	A equipe fica responsável por verificar as correções e esclarecimentos efetuados pelo PO, e registrar se estes foram suficientes ou se novos esclarecimentos e correções são necessários.
Participantes:	• PO
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• *User Stories* e/ou critérios de aceitação atualizados para esclarecer os problemas encontrados na verificação dos requisitos • *User Stories* e/ou critérios de aceitação atualizados com a correção dos critérios que não foram atendidos

Artefatos Produzidos:	• Registro indicando que os requisitos foram esclarecidos e/ou corrigidos a contento, ou; • Registro indicando que novos esclarecimentos e/ou correções são necessários.
Identificador	**UNR.AJGR.CONC.0003**
Nome:	Ajustar granularidade das *User Stories*
Descrição:	Este componente tem por propósito revisitar as *User Stories* e verificar as sugestões da equipe de desenvolvimento quanto às mudanças de granularidade das mesmas. Ao final de sua execução, o PO pode optar por acatar e implementar as sugestões da equipe, ou descartá-las. Caso, ao fim do componente anterior, os requisitos sejam avaliados como adequados pela equipe, a execução deste componente torna-se desnecessária. Por outro lado, caso sua execução se concretize, a equipe de desenvolvimento fica responsável por determinar as pessoas responsáveis pela execução do componente, bem como obter os recursos necessários a sua execução, de acordo com a configuração da equipe (co-localizada ou distribuída). As atividades compreendidas por este componente estão descritas na seção Arquitetura Interna a seguir.
Tipo de Componente:	Concreto
Obrigatório:	Não
Participantes:	• Equipe de desenvolvimento
Responsáveis	• PO

Critérios de Entrada	• Avaliação de requisitos finalizado
Critérios de Saída	• Sugestões de modificação aceitas ou descartadas pelo PO
Artefatos Requeridos:	• *User Stories*
Artefatos Produzidos:	• Notificação da equipe de desenvolvimento ao PO, sugerindo *User Stories* a serem agrupadas e/ou desmembradas; • *Backlog* atualizado com *User Stories* agrupadas ou desmembradas, ou; • Notificação do PO a equipe de desenvolvimento quanto ao descarte das sugestões;
Características Atendidas:	• *Iteration Planning*
Variantes deste Componente:	Não há
Arquitetura Interna:	
Atividade	**Sugerir desmembramento de *User Stories***
Descrição:	Caso a equipe identifique que a granularidade ou coesão de uma *User Story* é inadequada, pode sugerir seu desmembramento em outras stories, de forma a facilitar seu entendimento e permitir uma estimativa mais confiável ao PO. Ao final desta atividade, os participantes devem especificar como, em sua visão, as *User Stories* devem ser desmembradas de forma a obter granularidade e coesão adequados.

Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• *User Stories*
Artefatos Produzidos:	• Lista de *User Stories* com sugestões de desmembramento
Atividade	**Sugerir agrupamento de *User Stories***
Descrição:	Caso a equipe identifique que a granularidade ou coesão de um conjunto de *User Stories* é inadequada, pode sugerir seu agrupamento em uma só *User Story*. O objetivo do agrupamento, em geral, é facilitar seu entendimento e prover uma estimativa mais confiável ao PO. Os participantes ficam responsáveis por, ao final desta atividade, especificar como as *User Stories* devem ser agrupadas para prover melhor granularidade e coesão.
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• *User Stories*
Artefatos Produzidos:	• Lista de *User Stories* com sugestões de agrupamento
Atividade	**Avaliar sugestões**
Descrição:	Caso a equipe tenha produzido sugestões de agrupamento ou desmembramento de *User Stories*, o PO fica responsável por avaliar estas sugestões e decidir quais serão implementadas ou descartadas.

Participantes:	• Equipe de desenvolvimento
Responsáveis	• PO
Artefatos Requeridos:	• Lista de *User Stories* com sugestões de agrupamento. • Lista de *User Stories* com sugestões de desmembramento
Artefatos Produzidos:	• Sugestões de mudanças de granularidade que serão implementadas e/ou descartadas;
Atividade	**Implementar sugestões de mudança de granularidade nas *User Stories***
Descrição:	Caso o PO tenha aceito uma ou mais sugestões de agrupamento / desmembramento, o *Backlog* deve ser atualizado para refletir as sugestões.
Participantes:	• Equipe de desenvolvimento
Responsáveis	• PO
Artefatos Requeridos:	• Sugestões de mudança de granularidade a serem implementadas.
Artefatos Produzidos:	• *Backlog* atualizado com granularidade das *User Stories* modificada; • Notificação do PO a equipe de desenvolvimento quanto a finalização da implementação das mudanças sugeridas;
Identificador	**UNR.REES.CONC.0004**
Nome:	Refinar estimativas

Descrição:	Este componente tem por propósito revisitar e atualizar as estimativas que constam no *Backlog* de acordo com os esclarecimentos fornecidos pelo PO e novos agrupamentos/desmembramentos de *User Stories*. Caso não tenha havido necessidade de esclarecimentos de requisitos ou mudanças no *Backlog*, a execução deste componente se torna desnecessária.
Tipo de Componente:	Concreto
Obrigatório:	Não
Participantes:	• Equipe de desenvolvimento
Responsáveis	• PO
Critérios de Entrada	• Avaliação de requisitos finalizada • Esclarecimento de requisitos finalizado • Granularidade das *User Stories* ajustada
Critérios de Saída	• Estimativas atualizadas
Artefatos Requeridos:	• *Backlog* de funcionalidades
Artefatos Produzidos:	• *Backlog* atualizado refletindo a mudança de estimativas
Características Atendidas:	• *Iteration Planning*
Variantes deste Componente:	Não há

Arquitetura Interna:	Não há
Identificador	**UNR.FORE.ABST.0005**
Nome:	Fornecer requisitos da iteração à equipe
Descrição:	Este componente visa fornecer as *User Stories* e critérios de aceitação das funcionalidades a serem desenvolvidas na iteração. Isto pode ser feito de formas variadas, conforme as variantes deste componente. No entanto, independente da forma escolhida, este componente envolve: • Fornecer as *User Stories* que foram priorizadas, em conjunto com seu valor e os objetivos de negócio relacionados; • Fornecer os critérios de aceitação de cada funcionalidade, ilustrando os cenários de negócio a serem atendidos; Eventualmente haverá dúvidas ou pontos a serem esclarecidos acerca dos requisitos. Estas dúvidas devem ser registradas para posterior esclarecimento ou devem ser esclarecidas imediatamente pelo PO, de acordo com a variante concreta escolhida.
Tipo de Componente:	Abstrato
Participantes:	• Equipe de desenvolvimento
Responsáveis	• PO
Critérios de Entrada	• Requisitos aceitos pela equipe de desenvolvimento • *Backlog* da iteração definido
Critérios de Saída	• Requisitos fornecidos

Artefatos Requeridos:	• *Backlog* de funcionalidades • Requisitos
Artefatos Produzidos:	Não há
Características Atendidas:	• *Iteration Planning* • Fornecimento de requisitos na ausência do PO
Variantes deste Componente:	• UNR.APRE.CONC.0006 • UNR.DIRE.CONC.0007
Arquitetura Interna	Não há
Identificador	**UNR.APRE.CONC.0006**
Nome:	Apresentar requisitos da iteração à equipe
Descrição:	Nesta variante, o fornecimento dos requisitos se dá durante a reunião de planejamento da iteração (*Iteration planning meeting*), onde o PO fica responsável por apresentar a toda a equipe de desenvolvimento os requisitos das funcionalidades que foram priorizadas e validadas para serem trabalhadas ao longo da iteração que se inicia.

	Nesta variante, o PO fica responsável por providenciar os recursos e ferramentas necessários à execução do componente, de acordo com as características da equipe (co-localizada ou distribuída). Exemplos de recursos incluem salas de reunião, ferramentas de áudio/videoconferência, projetores etc.). Ao longo da reunião, o PO apresenta as *User Stories*, explicando seu valor e os objetivos de negócio que serão alcançados com a entrega daquela funcionalidade. Os critérios de aceitação também são apresentados, de forma a deixar a equipe ciente dos cenários de negócio a serem atendidos pela funcionalidade, e esclarecendo eventuais dúvidas. Nesta variante, se houver dúvidas ou pontos a serem esclarecidos, estes podem ser registrados em ferramenta previamente acordada ou esclarecidos imediatamente, dependendo da complexidade destes pontos.
Tipo de Componente:	Concreto
Obrigatório:	Não
Participantes:	• Equipe de desenvolvimento
Responsáveis	• PO
Critérios de Entrada	• Requisitos aceitos pela equipe de desenvolvimento • *Backlog* da iteração definido
Critérios de Saída	• Requisitos fornecidos
Artefatos Requeridos:	• *Backlog* de funcionalidades

	• Requisitos
Artefatos Produzidos:	Não há
Características Atendidas:	• *Iteration Planning*
Variantes deste Componente:	Não há
Arquitetura Interna:	Não há
Identificador	**UNR.DIRE.CONC.0007**
Nome:	Disponibilizar requisitos da iteração à equipe sem apresentação
Descrição:	Nesta variante, o fornecimento dos requisitos (descrição, valor de negócio, etc.) se dá por meio de ferramenta previamente acordada, não havendo apresentação por parte do PO. Mediante a disponibilização destas informações, a equipe de desenvolvimento fica responsável por registrar dúvidas e pontos a serem esclarecidos posteriormente pelo PO.
Tipo de Componente:	Concreto
Obrigatório:	Não
Participantes:	• Equipe de desenvolvimento
Responsáveis	• PO
Critérios de Entrada	• Requisitos aceitos pela equipe de desenvolvimento

	• *Backlog* da iteração definido
Critérios de Saída	• Requisitos fornecidos
Artefatos Requeridos:	• *Backlog* de funcionalidades • Requisitos • Informações de potencial de reuso
Artefatos Produzidos:	Não há
Características Atendidas:	• *Iteration Planning* • Fornecimento de requisitos na ausência do PO
Variantes deste Componente:	Não há
Arquitetura Interna:	Não há
Identificador	**UNR.FOIN.ABST.0008**
Nome:	Fornecer informações de potencial de reuso à equipe
Descrição:	O PO é responsável por fornecer informações que possam influenciar o projeto das funcionalidades do ponto de vista de reuso. Estas informações incluem, por exemplo: • sistemas que podem consumir/reutilizar a funcionalidade sendo desenvolvida; ou, • serviços existentes a serem reutilizados para apoiar a funcionalidade. Estes serviços podem ser utilizados na íntegra ou pode ser necessário adaptá-los para poderem ser utilizados;

	• fontes de dados externas a serem reutilizadas; Estas informações podem ser fornecidas à equipe de desenvolvimento de formas variadas, conforme os componentes concretos descritos a seguir.
Tipo de Componente:	Abstrato
Participantes:	• Equipe de desenvolvimento
Responsáveis	• PO
Critérios de Entrada	• Informações de potencial de reuso definidas
Critérios de Saída	• Informações de potencial de reuso fornecidas
Artefatos Requeridos:	• *Backlog* de funcionalidades • Informações de potencial de reuso
Artefatos Produzidos:	Não há
Características Atendidas:	• *Iteration Planning* • Composabilidade • Reuso • Fornecimento de informações de potencial de reuso na ausência do PO
Variantes deste Componente:	• UNR.APIN.CONC.0009 • UNR.DIRE.CONC.0010
Arquitetura Interna	Não há

Identificador	**UNR.APIN.CONC.0009**
Nome:	Apresentar informações de potencial de reuso à equipe
Descrição:	Nesta variante, as informações de potencial de reuso das funcionalidades priorizadas são apresentadas pelo PO a toda a equipe de desenvolvimento. O PO fica responsável por providenciar os recursos e ferramentas necessários à execução do componente, de acordo com a configuração da equipe. Em havendo dúvidas ou pontos a serem esclarecidos, estes podem ser registrados em ferramenta previamente acordada ou esclarecidos imediatamente, dependendo da complexidade destes pontos.
Tipo de Componente:	Concreto
Obrigatório:	Não
Participantes:	• Equipe de desenvolvimento
Responsáveis	• PO
Critérios de Entrada	• Informações de potencial de reuso definidas • *Backlog* da iteração definido
Critérios de Saída	• Informações de potencial de reuso fornecidas
Artefatos Requeridos:	• *Backlog* de funcionalidades • Informações de potencial de reuso
Artefatos Produzidos:	Não há

Características Atendidas:	• *Iteration Planning* • Composabilidade • Reuso
Variantes deste Componente:	Não há
Arquitetura Interna:	Não há
Identificador	**UNR.DIRE.CONC.0010**
Nome:	Disponibilizar informações de potencial de reuso sem apresentação
Descrição:	Nesta variante, o fornecimento das informações de potencial de reuso se dá por meio de ferramenta previamente acordada, não havendo apresentação por parte do PO. Mediante a disponibilização destas informações, a equipe de desenvolvimento fica responsável por registrar dúvidas e pontos a serem esclarecidos, para posterior esclarecimento pelo PO.
Tipo de Componente:	Concreto
Obrigatório:	Não
Participantes:	• Equipe de desenvolvimento
Responsáveis	• PO
Critérios de Entrada	• Informações de potencial de reuso definidas • *Backlog* da iteração definido
Critérios de Saída	• Informações de potencial de reuso fornecidas

Artefatos Requeridos:	• *Backlog* de funcionalidades • Informações de potencial de reuso
Artefatos Produzidos:	Não há
Características Atendidas:	• *Iteration Planning* • Composabilidade • Reuso • Fornecimento de informações de potencial de reuso na ausência do PO
Variantes deste Componente:	Não há
Arquitetura Interna:	Não há
Identificador	**UNR.REFU.CONC.0012**
Nome:	Repriorizar funcionalidades
Descrição:	De forma análoga ao componente UNR.ESRE.CONC.0002, pode haver dúvidas ou esclarecimentos de complexidade tal que exceda a capacidade do PO em produzir esclarecimentos em tempo hábil. Por exemplo, em casos onde o PO representa um conjunto de clientes, pode ser necessário alinhar interesses ou obter maiores informações sobre o negócio para responder alguma dúvida, excedendo o tempo que a equipe pode esperar por esclarecimentos.

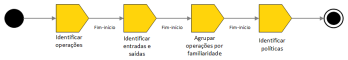
	Neste caso, o PO pode optar pela repriorização destas funcionalidades para alguma iteração subsequente, caso necessite de mais tempo para obter respostas. Esta repriorização deverá levar em consideração a velocidade estimada da equipe.
Tipo de Componente:	Concreto
Obrigatório:	Não
Participantes:	• Equipe de desenvolvimento
Responsáveis	• PO
Critérios de Entrada	• Há pontos pendentes de esclarecimentos • Não há tempo hábil para esclarecer os pontos
Critérios de Saída	• Funcionalidades repriorizadas
Artefatos Requeridos:	• Pontos pendentes de esclarecimentos
Artefatos Produzidos:	• *Backlog* atualizado com mudanças de prioridades
Características Atendidas:	• *Iteration Planning*
Variantes deste Componente:	Não há
Arquitetura Interna:	Não há
Identificador	**UNR.IDME.CONC.0013**
Nome:	Identificar metáforas

Descrição:	Segundo KEELING & VELICHANSKY [68] o uso de Metaphors (ou Metáforas) provê um direcionamento para a tomada de decisões de projeto. Em seu experimento, uma Metáfora bem definida ajudou o time a tomar decisões arquiteturais importantes e guiou a implementação de funcionalidades de baixo nível. Generalizando para este princípio, uma Metáfora bem estabelecida pode oferecer direcionamento tanto para decisões arquiteturais de separação entre contrato, implementação e consumidores de um serviço, quanto para a construção e refatoração destes. Se uma dada Metáfora for incorporada como um termo de negócio no código, a equipe deve registrar seu significado em local acordado com o PO, de forma a possibilitar seu entendimento por pessoas que venham a se juntar à equipe de desenvolvimento em momento futuro. Não há um conjunto de passos a serem seguidos que sempre resulte em boas metáforas **(a)**. KEELING & VELICHANSKY [68] sugerem formas de se identificar uma Metáfora, estando a maioria relacionada a particularidades de um projeto, ou situações vividas por um time.
Tipo de Componente:	Concreto
Obrigatório:	Sim
Participantes:	• PO
Responsáveis	• Equipe de desenvolvimento

Critérios de Entrada	• Requisitos fornecidos
Critérios de Saída	• Requisitos revisados para identificação de metáforas
Artefatos Requeridos:	• Requisitos
Artefatos Produzidos:	• Metáforas do projeto
Características Atendidas:	• *Iteration Planning*
Variantes deste Componente:	Não há
Arquitetura Interna:	Não há
Identificador	**UNR.AUTE.CONC.0014**
Nome:	Automatizar testes de aceitação
Descrição:	Uma vez definidas as metáforas que guiam o entendimento do negócio, o passo seguinte é transformar os critérios de aceitação em testes de aceitação. Os critérios de aceitação, funcionais e não-funcionais **(a)**, apresentados no início da iteração e escritos de forma textual, devem ser traduzidos para código executável, chamados de testes de aceitação, conforme praticado por KARSTEN & CANNIZZO [15]. Existem diferentes ferramentas para apoiar esta tarefa **(b)**.

	Conforme WAKE [56] e BECK [16], o *Design* é construído de forma iterativa e incremental, como a resposta mais simples que satisfaz a suíte de testes, dentre os quais estão os critérios de aceitação. O *Design* do código é um produto final, e não inicial. Ele expressa o mínimo necessário e suficiente para atender aos requisitos de negócio aderente à prática *Simple Design* do XP **(c)**.
Tipo de Componente:	Concreto
Obrigatório:	Sim
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Critérios de Entrada	• Requisitos revisados para identificação de metáforas
Critérios de Saída	• Testes de aceitação automatizados
Artefatos Requeridos:	• Requisitos
Artefatos Produzidos:	• Testes de aceitação
Características Atendidas:	• Testes de aceitação
Variantes deste Componente:	Não há
Arquitetura Interna:	Não há
Identificador	**UNR.PRCO.CONC.0015**
Nome:	Projetar contrato mínimo

Descrição:	Conforme definido por ERL [4], um serviço deve expressar suas capacidades, restrições e informações semânticas por meio de um contrato capaz de ser interpretado por máquinas. Para atingir este objetivo, ERL [4] propõe a implantação de padrões de governança das políticas dos serviços, bem como da representação dos dados trafegados por estes. No entanto, de acordo com KROGDAHL, LUEF, & STEINDL [10], não é possível ter todos os detalhes de um contrato no início do desenvolvimento, especialmente em cenários onde a incerteza é alta. Então, KROGDAHL, LUEF, & STEINDL [10] recomendam que os detalhes do contrato sejam deixados em aberto e incrementados conforme necessário, ao invés de tentar supor todos os detalhes no início. Desta forma, o conjunto de serviços estará sempre aderente as necessidades do negócio. Assim, o contrato, contemplando políticas e as representações de seus dados deve ser atualizado de forma iterativa e incremental, de acordo com as necessidades bem definidas (presentes ou futuras) de negócio naquele momento **(a)**, sendo refatorado continuamente conforme a necessidade se apresenta **(b)**. A seção Arquitetura Interna a seguir define as atividades a serem desempenhadas para atingir os objetivos descritos.
Tipo de Componente:	Concreto
Obrigatório:	Sim
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento

Critérios de Entrada	• Requisitos fornecidos
Critérios de Saída	• Contrato mínimo identificado
Artefatos Requeridos:	• Requisitos
Artefatos Produzidos:	• Contrato mínimo da funcionalidade
Características Atendidas:	• Contrato
Variantes deste Componente:	Não há
Arquitetura Interna:	
Atividade	**Identificar operações**
Descrição:	Para atingir o objetivo de expressar as capacidades de um serviço, a equipe deve, primeiramente, identificar as suas operações, conforme definido por ERL [4]. Os testes de aceitação e *User Stories* servem de apoio nesta atividade. O entendimento das ações a serem invocadas por um usuário sobre um serviço, bem como das entradas e saídas esperadas, servem de ponto de partida para identificar as operações a serem suportadas por um dado serviço. Pode ser aplicado também algum método de identificação de operações, como, o apresentado por AZEVEDO *et al.* [69]. Uma vez identificadas as interações, o passo seguinte é identificar os os dados trafegados por essas interações.
Participantes:	• Equipe de desenvolvimento

Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Requisitos
Artefatos Produzidos:	• Contrato do serviço.
Atividade	**Identificar entradas e saídas**
Descrição:	Uma vez identificadas as operações, a equipe deve, a seguir, identificar entradas e saídas. Os testes de aceitação e *User Stories* mais uma vez servem de apoio. Estes identificam informações importantes a serem trocadas entre provedor e consumidor para atingir os objetivos de negócio. Estas informações definem o conjunto inicial de entradas e saídas de um serviço. Conforme o entendimento da funcionalidade evolui, outras informações são identificadas e podem ser agrupadas logicamente, dando origem a mensagens mais complexas, passíveis de reutilização por outras funcionalidades ou serviços.
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Requisitos
Artefatos Produzidos:	• Contrato do serviço.
Atividade	**Agrupar operações por afinidade**

Descrição:	A evolução do entendimento dos propósitos de negócio, bem como a evolução do próprio projeto, trazem a criação de novas operações ou agrupamento de outras de acordo com os contextos funcionais e não-funcionais que visam atender. Uma vez definido um grupo de operações candidatas, estas devem ser agrupadas por afinidade. Para tal fim, pode ser aplicado algum método de identificação de serviços e/ou operações, como, por exemplo, o apresentado por DIIRR *et al.* [54], a fim de agrupar operações afins. Grupos estes que, posteriormente, darão origem a serviços em si.
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Requisitos
Artefatos Produzidos:	• Contrato do serviço.
Atividade	**Identificar políticas**
Descrição:	Conforme definido por ERL [4], um serviço deve expressar, entre outros itens, suas políticas, de forma a permitir a avaliação por humanos e máquinas de suas capacidades. Para cumprir tal objetivo, a equipe deve identificar estas políticas no contrato do serviço. Tipicamente, testes de aceitação não-funcionais são um bom ponto de partida para a identificação inicial de políticas. Por exemplo, padrões de segurança corporativos podem ditar políticas de segurança do serviço. Padrões de utilização de rede, por sua vez, podem definir políticas de utilização do serviço.

Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Requisitos
Artefatos Produzidos:	• Contrato do serviço.
Identificador	**UNR.PRBA.CONC.0016**
Nome:	Projetar baixo acoplamento
Descrição:	De acordo com ERL [4], um serviço é fracamente acoplado se as dependências entre contrato, implementação e consumidores são mínimas. Analogamente, segundo NARAYANAN [70], a lógica do serviço deve ser independente de produtos, tecnologias ou protocolo de transporte. Esta não deve se preocupar, tampouco, com interesses ortogonais, como *log*, tratamento de erros, métricas e segurança. A lógica do serviço deve estar relacionada única e exclusivamente com a capacidade de negócio a ser atendida. ERL [4] define diversas medidas de acoplamento, as quais serão endereçadas pelas diversas atividades de *Design* definidas por este componente com o propósito de auxiliar no projeto de serviços e funcionalidades para atender ao princípio de baixo acoplamento.

	Complementarmente, a fim de atender a esta definição, NARAYANAN [70] sugere o uso iterativo de práticas do XP, como *Simple Design* e *Refactoring*, aliado aos tipos de acoplamento propostos por ERL [4] **(a)**. O *Simple Design* atua minimizando preocupações excessivas com ambiente e consumidores, maximizando o foco na lógica de negócio. Em conjunto, *Refactoring* é usado ao longo do tempo para diminuir o acoplamento entre serviço e aquilo que o circunda. A seção Arquitetura Interna apresenta as atividades a serem executadas pela equipe para atingir estes objetivos.
Tipo de Componente:	Concreto
Obrigatório:	Sim
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Critérios de Entrada	• Requisitos fornecidos
Critérios de Saída	• Contrato atualizado com informações de baixo acoplamento
Artefatos Requeridos:	• Requisitos
Artefatos Produzidos:	• Contrato do serviço • Testes automatizados de aceitação
Características Atendidas:	• Baixo acoplamento

Variantes deste Componente:	Não há
Arquitetura Interna:	
Atividade	**Projetar acoplamento da lógica ao contrato**
Descrição:	Segundo ERL [4], o acoplamento da lógica ao contrato se refere a ter uma lógica de serviço aderente ao que é exposto no contrato. Assim, o projeto de um serviço deve procurar maximizar este tipo de acoplamento. Conforme ERL [4], isto pode ser atingido seguindo uma abordagem *contract-first*, a qual é endereçada pela execução do componente UNR.PRCO.CONC.0015. Com a definição prévia do contrato mínimo, devem ser criados testes automatizados de aceitação, os quais farão o tratamento do acoplamento da lógica ao contrato. Estes testes exercitarão o funcionamento conjunto de todas as camadas e componentes que colaboram para atender a uma necessidade de negócio, verificando, assim, se a funcionalidade responde conforme definido pelo contrato.
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Requisitos
Artefatos Produzidos:	• Contrato do serviço • Testes automatizados de aceitação
Atividade	**Projetar acoplamento do contrato à lógica**

Descrição:	O acoplamento do contrato à lógica se refere a evitar que a definição do contrato esteja presa a lógica interna do serviço. Se isto ocorrer, o impacto a consumidores será grande no caso de mudanças, uma vez que estes estarão acoplados a algo com tendência a mudar com frequência (lógica de negócio) ao invés de estarem acoplados a algo que tende a estabilidade (interfaces e contratos). Segundo ERL [4], por estas razões acima, este tipo de acoplamento deve ser minimizado. Uma forma de endereçá-lo é a abordagem *contract-first* [71], discutida no componente anterior e endereçada pela execução do componente UNR.PRCO.CONC.0015. Outra forma de evitá-lo, segundo ERL [4], é evitar que o contrato seja automaticamente derivado de artefatos pré-existentes, como, por exemplo, *schemas* XSD. De forma análoga ao componente anterior, testes automatizados de aceitação também devem ser criados a fim de minimizar este tipo de acoplamento. Conforme o projeto evolui e o código é modificado, estes testes, que interagem primariamente com o contrato, servem de apoio para identificar quebras de compatibilidade no contrato ocasionadas por uma mudança de lógica.
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Requisitos • Contrato do serviço

Artefatos Produzidos:	• Contrato do serviço atualizado • Testes automatizados de aceitação
Atividade	**Projetar acoplamento do contrato à tecnologia**
Descrição:	O acoplamento do contrato à tecnologia diz respeito à exposição no contrato de detalhes que forcem o consumidor a conhecer tecnologias ou protocolos proprietários para interagir com o serviço. Este tipo de acoplamento deve ser minimizado [4]. Para abordar este tipo de acoplamento, deve-se verificar se o contrato expõe detalhes da plataforma sobre a qual executa (por exemplo, Java ou .Net) ou define o uso de protocolos proprietários para a comunicação (por exemplo, algum protocolo de segurança provido por terceiros) em detrimento de padrões abertos como WS-* ou OAuth.
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Requisitos • Contrato do serviço
Artefatos Produzidos:	• Contrato do serviço atualizado
Atividade	**Projetar acoplamento do contrato à implementação**

Descrição:	Segundo ERL [4], para atender a uma requisição de negócio, qualquer serviço deverá internamente interagir com produtos ou tecnologias diversas, como, por exemplo, SGBD's, API's de sistemas legados, estruturas de segurança, diretórios contidos em servidores, entre outros.No entanto, estes detalhes devem permanecer ocultos do consumidor, haja vista que, caso o fornecedor de SGBD mude, por exemplo, o consumidor não deve ser impactado nem forçado a usar uma nova versão do contrato com o serviço. Para endereçar este tipo de acoplamento deve-se verificar se o contrato expõe detalhes da plataforma sobre a qual executa, componentes utilizados internamente, caminhos de arquivos etc., e eliminar/minimizar tais referências.
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Requisitos • Contrato do serviço
Artefatos Produzidos:	• Contrato do serviço atualizado
Atividade	**Projetar acoplamento do contrato ao escopo funcional**
Descrição:	O acoplamento do contrato ao escopo funcional ocorre quando um contrato é criado para suportar um escopo restrito de consumidores. Situações onde isso se passa incluem serviços que implementam processos de negócio específicos ou serviços que estabelecem comunicação com um parceiro externo por meio de uma assinatura específica [4].

	Apesar de indesejável, este tipo de acoplamento pode ser inevitável. Por exemplo, em cenários de B2B, se um serviço é criado com o propósito específico de se conectar a um parceiro de negócio, pode não ser viável torná-lo mais genérico com o propósito de aumentar seu reuso, sob pena de prejudicar a comunicação com este parceiro de negócios. Para evitar este tipo de acoplamento, deve-se verificar se não há sobreposição de escopo entre serviços de uma organização.De forma geral, a minimização deste tipo de acoplamento passa também por questões de medidas de reuso, tratadas mais a frente nesta LPS.
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Requisitos • Contrato do serviço
Artefatos Produzidos:	• Contrato do serviço atualizado
Atividade	**Projetar conexões físicas**
Descrição:	KRAFZIG [29] define duas medidas para acoplamento por conexões físicas: ponto a ponto ou via mediador, apresentando o último menor acoplamento do que o primeiro. Para abordar esta questão, a equipe deve decidir pelo tipo de conexão física mais adequada ao seu contexto.

	A escolha da forma de conexão física pode não ser uma decisão da equipe. Por exemplo, se a organização onde o serviço será implantado não dispuser de mecanismos de mediação (por exemplo, barramento de serviços), só será possível fazer uso de conexões ponto a ponto, ainda que sob pena de aumentar o acoplamento da solução. Por outro lado, se há um mecanismo de mediação estabelecido pela organização, e uma política de utilização do mesmo, então esta informação deverá constar no critério de aceitação em forma de requisito não-funcional.
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Requisitos • Contrato do serviço
Artefatos Produzidos:	• Definição do tipo de conexões físicas a ser utilizado na implementação do serviço
Atividade	**Projetar estilo de comunicação**
Descrição:	Conforme KRAFZIG [29], há dois tipos possíveis de acoplamento relacionados ao estilo de comunicação: síncrona ou assíncrona. Comunicação assíncrona leva a menor acoplamento. Para abordar este tipo de acoplamento, a equipe deve decidir pelo estilo de comunicação ótimo em relação às suas necessidades. Há casos, no entanto, em que esta definição pode não ser uma decisão da equipe, pois permeia questões de desempenho, experiência de usuário e governança.

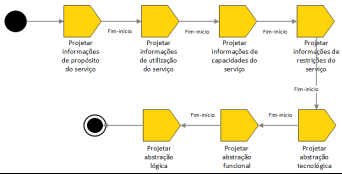
	Por exemplo, se o desempenho do serviço se deteriora de acordo com os parâmetros da requisição, mas essa deterioração produz respostas em um tempo aceitável para o PO sem comprometer a disponibilidade do serviço, então o PO pode fazer constar nos requisitos funcionais que a comunicação deve ser síncrona. Analogamente, se houver estabelecida uma diretriz onde os serviços devem preferencialmente ser assíncronos, esta deve constar dos critérios de aceitação. O serviço produzido será sempre a forma mais simples de atender aos critérios de aceitação.
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Requisitos • Contrato do serviço
Artefatos Produzidos:	• Definição do estilo de comunicação a ser utilizado na implementação do serviço
Atividade	**Projetar estruturas de dados**
Descrição:	Segundo KRAFZIG [29], a discussão sobre o nível de acoplamento trazido por estruturas de dados gira em torno do uso de tipos de complexos ou simples, devendo a equipe decidir pelo tipo de estrutura mais adequada à sua realidade.

	Quando se fala de tipos, segundo JOSUTTIS [3], intrinsecamente está se falando de harmonização de tipos em uma corporação. É comum uma organização ter dificuldades em garantir que todos os seus sistemas usem o mesmo modelo canônico de objetos de negócio. Um modelo canônico de objetos de negócio é um modelo onde as informações de negócio são modeladas de uma forma que é semantica e estruturalmente aderente a um conjunto de regras definidas para possibilitar a comunicação entre aplicações ou partes da organização [72]. As dificuldades de uma organização em garantir a utilização do mesmo modelo canônico vem dos fatos de: (i) os seus sistemas evoluírem em tempos diferentes; e (ii) os responsáveis por cada sistema terem uma visão diferente sobre um conceito do negócio. Em combinação, estes fatores levam a sistemas que se comunicam por meio de representações diferentes do mesmo objeto de negócio requerendo o uso de transformação de dados para efetivar a comunicação. Assim, o acoplamento por estruturas de dados é otimizado trabalhando com as estruturas mais simples possíveis, permitindo menor acoplamento entre os sistemas de uma corporação. Note-se que isto não significa dar preferência a tipos simples (por exemplo, *String*, *Integer*, *Boolean*) como parâmetros de serviços. Ao contrário, esta recomendação diz respeito a não dedicar muito esforço na tentativa de se definir um modelo de comunicação complexo, tentando prever e contemplar todas as visões possíveis sobre um objeto de negócio.
Participantes:	• Equipe de desenvolvimento

Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Requisitos • Contrato do serviço
Artefatos Produzidos:	• Definição das estruturas de dados a serem utilizadas na implementação do serviço
Atividade	**Projetar tipagem**
Descrição:	Segundo KRAFZIG [29], favorece-se um menor acoplamento ao trabalhar com tipos de dados fracos em detrimento de tipos fortes. A equipe deve decidir a tipagem mais adequada ao seu contexto. Tipos fortes são aqueles que requerem validação de sua integridade. Por exemplo, se uma dada informação é definida como numérica, não deve ser possível atribuir a ela o valor "essa é uma *string*". Analogamente, uma informação fracamente tipada é mais genérica e dispensa validação de seu conteúdo. Conforme JOSUTTIS [3], tipos fortes são de mais fácil processamento e mais fácil de se trabalhar pelo programador. Por outro lado, uma informação fortemente tipada precisa ser validada (no caso de XML, a validação é feita via *schemas* XSD), o que demanda tempo de processamento. Não há uma recomendação sobre a melhor forma de trabalhar independente de contexto. A tipagem deverá ser definida pela equipe como a resposta mais simples aos critérios de aceitação, levando em consideração questões como tempo de processamento, acoplamento, e evolução.
Participantes:	• Equipe de desenvolvimento

Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Requisitos • Contrato do serviço
Artefatos Produzidos:	• Definição da tipagem a ser utilizada na implementação do serviço
Atividade	**Projetar padrão de interação**
Descrição:	"Padrão de interação", segundo KRAFZIG [29], diz respeito ao uso de árvores de objetos complexos ou mensagens auto-contidas (*self-contained messages*). A equipe deve decidir aquele que melhor atende aos requisitos de seu projeto. O conceito de *self-contained messages*, está relacionado à complexidade de objetos. Segundo JOSUTTIS [3], quanto mais complexa a definição de um objeto de negócio e quanto menos tipos de dados fundamentais (por exemplo, *String*, *Integer*, *Boolean*) são usados em sua definição, maior o risco de se deparar com problemas de portabilidade. Por exemplo, o uso do tipo *List* em detrimento de *arrays*, em serviços construídos em Java, tipicamente representa um risco de portabilidade. Linguagens como, por exemplo, C#, frequentemente tem dificuldades de interagir com coleções que representam List, ao passo que o uso de array para coleções é aceito nativamente. JOSUTTIS [3] recomenda que, de forma geral, objetos customizados sejam endereçados de forma conservadora, preferindo objetos menos complexos e que façam uso extensivo de tipos de dados fundamentais, gerando mensagens auto-contidas.
Participantes:	• Equipe de desenvolvimento

Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Requisitos • Contrato do serviço
Artefatos Produzidos:	• Definição do padrão de interação a ser utilizado na implementação do serviço
Identificador	**UNR.PRAB.CONC.0017**
Nome:	Projetar abstração
Descrição:	Conforme ERL [4], o princípio de abstração diz respeito a omitir do contrato informações que detalhem o funcionamento interno do serviço, dando preferência a informações sobre como utilizar o serviço, suas capacidades, restrições e propósito de suas operações. Este componente especifica atividades que visam considerar questões relativas à exposição de informações sobre capacidades, restrições e propósito de negócio do serviço, descrevendo tantos aspectos funcionais quanto não-funcionais. Em complemento, há atividades que tratam das medidas de abstração propostas por ERL [4]: tecnológica, lógica e funcional e qualidade de serviço.
Tipo de Componente:	Concreto
Obrigatório:	Sim
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Critérios de Entrada	• Requisitos fornecidos

Critérios de Saída	• Abstração projetada
Artefatos Requeridos:	• Requisitos
Artefatos Produzidos:	• Contrato atualizado com informações de abstração
Características Atendidas:	• Abstração
Variantes deste Componente:	Não há
Arquitetura Interna:	
Atividade	**Projetar informações de propósito do serviço**
Descrição:	Esta atividade trata da exposição dos propósitos de um serviço. A equipe deve desempenhar o projeto do contrato, levando em consideração informações como: • Problemas de negócio solucionados pelo serviço; • Consumidores primariamente beneficiados pelo serviço. Tipicamente, informações sobre propósito expõem "a quê" um serviço se propõe, bem como "a quem".
Participantes:	• PO
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Requisitos

Artefatos Produzidos:	• Contrato atualizado com informações relativas ao propósito do serviço
Atividade	**Projetar informações de utilização do serviço**
Descrição:	Este componente endereça questões relativas a políticas de utilização do serviço. A equipe deve projetar um contrato para otimizar sua abstração, a partir de informações como: • SLA (*Service Level Agreement*) de disponibilidade ou períodos programados de indisponibilidade; • Forma de interação com o serviço (síncrono ou assíncrono); • Políticas de tratamento de carga (por exemplo, invocações que resultem em mais de 5s de execução em horários de pico serão retornadas assincronamente por e-mail). Informações como essas instruem o consumidor quanto ao que esperar do serviço, sem expor detalhes de implementação do mesmo.
Participantes:	• PO
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Requisitos
Artefatos Produzidos:	• Contrato atualizado com informações relativas a utilização do serviço
Atividade	**Projetar informações de capacidades do serviço**
Descrição:	Este componente endereça questões relativas a capacidades do serviços. Ao executá-lo, a equipe deve projetar o contrato para otimizar sua abstração, expondo informações como, por exemplo:

	• Processo de negócio atendido; • Área de negócio representada. De forma geral, informações sobre capacidades expõem "o quê" um serviço oferece sobre uma organização, ao invés de "como" o faz.
Participantes:	• PO
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Requisitos
Artefatos Produzidos:	• Contrato atualizado com informações relativas a capacidades do serviço
Atividade	**Projetar informações de restrições do serviço**
Descrição:	Para que um consumidor esteja plenamente ciente do que um serviço pode oferecer, é importante ressaltar também os seus requisitos inversos, ou seja, aquilo que não será oferecido. A equipe deve atualizar o projeto do contrato com informações sobre suas restrições, as quais podem incluir: • Processos de negócio relacionados que não são contemplados; • Informações sobre *timeout* de requisições; • Protocolos de segurança aceitos.
Participantes:	• PO
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Requisitos

Artefatos Produzidos:	• Contrato atualizado com informações relativas a restrições do serviço
Atividade	**Projetar abstração tecnológica**
Descrição:	Detalhes tecnológicos não devem ser considerados no contrato, tais como: • Linguagem de programação utilizada; • Recursos de *hardware* utilizados; • URL do servidor de banco de dados. Ao omitir estas informações, o serviço pode ser evoluído sem impacto visível ao consumidor, desde que o contrato permaneça estável. Na execução desta atividade, a equipe deve, então, remover detalhes tecnológicos destes tipos do projeto do contrato, de forma a otimizar sua abstração tecnológica.
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Requisitos
Artefatos Produzidos:	• Contrato atualizado para considerar abstração tecnológica
Atividade	**Projetar abstração lógica**
Descrição:	A equipe de desenvolvimento deve atualizar o contrato, minimizando informações como: • Algoritmo de ordenação utilizado; • Mecanismo de tratamento de exceções.
Participantes:	• Equipe de desenvolvimento

Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Requisitos
Artefatos Produzidos:	• Contrato atualizado para considerar abstração lógica
Atividade	**Projetar abstração funcional**
Descrição:	Abstração funcional diz respeito a ocultar do consumidor informações sobre operações internas do serviço, expondo somente aquilo que interessa ao consumidor externo. Por exemplo, se duas operações "públicas" internamente fazem uso de um mesmo método para ordenar clientes por data de contratação, o princípio de abstração funcional dita que este método não deve ser exposto, pois não se dispõe a uso por consumidores externos, mas somente a uso interno do serviço. De forma geral, ao executar este componente, a equipe deve garantir que o contrato exponha somente as operações que devem ser visíveis a um consumidor externo, evitando expor lógicas aplicáveis somente a procedimentos internos.
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Requisitos
Artefatos Produzidos:	• Contrato atualizado para considerar abstração funcional
Identificador	**UNR.PRRE.CONC.0018**
Nome:	Projetar reuso

Descrição:	Este princípio de reuso está atrelado ao posicionamento de serviços como ativos corporativos, estando sujeitos a demandas por retorno sobre o investimento feito. Assim, quanto mais reutilizado for um serviço, de forma direta ou por meio de reuso como parte de uma solução, maior será o retorno gerado sobre o investimento feito em seu desenvolvimento. De fato, segundo OLIVEIRA, RAMOS & DIAS JUNIOR [73], maximização de retorno de investimento é um dos fatores a exercer maior influência sobre reuso em SOA. Neste passo, as informações de potencial de reuso providas pelo PO no componente UNR.FOIN.ABST.0008 têm influência direta na forma como um serviço será projetado, com implicações sobre a complexidade do *Design* do serviço. Ainda segundo OLIVEIRA, RAMOS & DIAS JUNIOR [73], o projeto dos serviços é o momento mais influenciado pelo reuso um ciclo de desenvolvimento de software, de acordo com os trabalhos analisados em sua pesquisa. De acordo com o princípio de *Simple Design*, o projeto do serviço será não só a resposta mais simples que atende aos seus testes de aceitação, mas também aquela que atende a medida de reuso esperada **(a)**. De forma geral, a equipe de desenvolvimento, ao executar este passo, deve levar em consideração os seguintes aspectos durante o desenvolvimento de um serviço: • Qual a forma mais simples deste serviço atender às necessidades da organização (ilustrados pelos critérios de aceitação)?

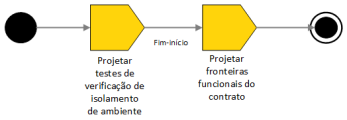
	• Esta forma mais simples atende à medida de reuso atrelada a este serviço?
Tipo de Componente:	Concreto
Obrigatório:	Sim
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Critérios de Entrada	• Requisitos fornecidos
Critérios de Saída	• Reuso projetado
Artefatos Requeridos:	• Requisitos • Testes de aceitação
Artefatos Produzidos:	• Definição sobre conectores, adaptadores, conversões, etc., para favorecer o reuso • Definição sobre medida de reuso do serviço • Definição sobre impactos a modelos canônicos • Notificação a consumidores sobre nova versão de contrato
Características Atendidas:	• Reuso
Variantes deste Componente:	Não há
Arquitetura Interna:	
Atividade	**Definir medida de reuso**

Descrição:	ERL [4] define três medidas de reuso, com respectivos impactos na complexidade do *Design* do serviço: • Tático: serviços nesta categoria se concentram nos requisitos funcionais imediatos; • Direcionado: nesta categoria, os serviços atendem às necessidades conhecidas, imediatas ou futuras; • Completo: serviços sob esta classificação, além de atender às necessidades conhecidas, visam contemplar possíveis necessidades futuras, sem que haja nenhuma definição destas. Ao executar esta atividade, a equipe deve decidir qual será a medida de reuso do serviço, de acordo com os critérios e informações de potencial de reuso fornecidas pelo PO. De forma geral, o princípio de *Simple Design* dita que o *design* deve ser a resposta mais simples aos testes de aceitação e que atenda às necessidades conhecidas, imediatas ou futuras. Assim, deve ser dada preferência ao reuso direcionado, de forma a não demandar constante refatoração dos serviços, nem requerer esforço extraordinário para implementar funcionalidades que podem nunca vir a serem consumidas.
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Requisitos
Artefatos Produzidos:	• Definição sobre medida de reuso do serviço

Atividade	**Projetar conectores / adaptadores**
Descrição:	Durante a apresentação dos requisitos, o PO tem a responsabilidade de fornecer informações relativas ao reuso. Uma destas informações diz respeito a serviços ou fontes de dados pré-existentes ou legados a serem reutilizados pela funcionalidade em desenvolvimento. No entanto, estes serviços ou fontes de dados a serem reutilizados podem requerer formas específicas de interação. Assim, a equipe deve projetar as conversões e adaptações que o novo serviço terá que contemplar para interagir com este ativo legado **(a)**.
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Requisitos
Artefatos Produzidos:	• Definição sobre conectores, adaptadores, conversões, etc., para favorecer o reuso
Atividade	**Projetar modificações em modelos canônicos**
Descrição:	Em organizações onde o negócio e as mensagens sejam modelados por meio de modelos canônicos, uma vez que o contrato inicial tenha sido definido, a equipe deve projetar os impactos sobre um modelo canônico e notificar demais consumidores dos serviços da organização, de forma a não prejudicar o andamento da iteração e permitir que as demais equipes possam planejar sua adaptação ao novo modelo.
Participantes:	• Equipe de desenvolvimento

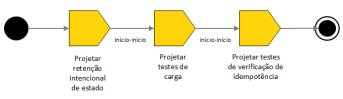
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Requisitos
Artefatos Produzidos:	• Definição sobre impactos a modelos canônicos
Atividade	**Notificar consumidores sobre nova versão de contrato**
Descrição:	No caso de modificações sobre serviços pré-existentes e já disponibilizados a consumidores, a equipe deve notificar esses consumidores sobre as modificações planejadas de forma que possam se planejar adequadamente para uma transição suave para a nova versão. Essa notificação deve incluir não somente informações funcionais / técnicas, como também informações sobre atualizações na documentação do serviço.
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Requisitos
Artefatos Produzidos:	• Notificação a consumidores sobre nova versão de contrato
Identificador	**UNR.PRAU.CONC.0019**
Nome:	Projetar autonomia
Descrição:	Este princípio, conforme descrito por ERL [4], diz respeito à delimitação lógica e física entre um serviço e seu ambiente de execução, e a delimitação funcional entre um serviço e demais integrantes de um repositório de serviços.

	De forma similar ao princípio de Baixo Acoplamento, *Simple Design* visa produzir as lógicas de serviços mais simples possíveis que atendem aos testes de aceitação, endereçando a **autonomia de tempo de execução** pela minimização de preocupações excessivas com ambiente e dando maior ênfase à lógica de negócio, conforme recomendado por NARAYANAN [70] **(a)**. Além disso, práticas de XP também possibilitam endereçar a **autonomia de tempo de *design***. O uso de *Coding Standards* influencia na criação de contratos que expõem com maior clareza suas fronteiras funcionais. Estas fronteiras funcionais podem ser definidas de forma mais explícita em cooperação com o *On-Site Customer* que conhece melhor os critérios de aceitação não-funcionais, as informações de potencial de reuso do serviço sendo desenvolvido, e, por conseguinte, a classificação do serviço quanto à sua **medida de autonomia**. A seção Arquitetura Interna apresenta as atividades a serem desempenhadas pela equipe para projetar um serviço com autonomia otimizada.
Tipo de Componente:	Concreto
Obrigatório:	Sim
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Critérios de Entrada	• Reuso projetado
Critérios de Saída	• Requisitos fornecidos

Artefatos Requeridos:	• Requisitos
Artefatos Produzidos:	• Contratos atualizados para posicionar operações de acordo com autonomia de tempo de *design* • Testes automatizados
Características Atendidas:	• Autonomia
Variantes deste Componente:	Não há
Arquitetura Interna:	
Atividade	**Projetar testes de verificação de isolamento de ambiente**
Descrição:	Testes automatizados e um ambiente de integração contínua são úteis para verificar se o isolamento entre um serviço e seu ambiente está adequado. Assim, a equipe deve criar testes que verifiquem se um serviço está adequadamente isolado de seu ambiente de execução. Por exemplo, no caso de funcionalidades que fazem acesso a recursos de hardware, testes podem ser projetados para verificar que este acesso é dinâmico, aceitando o ambiente de execução corrente e não um ambiente específico. Isto favorece a autonomia de tempo de execução, definida por ERL [4]. Além disso, num ambiente de integração contínua onde estes testes sejam executados frequentemente, caso este isolamento seja violado, a equipe será imediatamente notificada.
Participantes:	• Equipe de desenvolvimento

Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Requisitos
Artefatos Produzidos:	• Testes automatizados
Atividade	**Projetar fronteiras funcionais do contrato**
Descrição:	A definição de fronteiras funcionais está bastante relacionada a definições de entradas/saídas/afinidades, e questões de acoplamento, endereçados por meio dos componentes UNR.PRCO.CONC.0015 e UNR.PRBA.CONC.0016. Tipicamente, quando estes componentes forem previamente executados, esta atividade poderá ser dispensada, por tratarem do mesmo tópico. Em casos onde, por alguma razão, a pré-execução destes componentes não ocorra, a equipe deve verificar os contratos de serviços pré-existentes e dos serviços sendo desenvolvidos, de forma a identificar se não há sobreposição funcional entre os contratos. Ou seja, se um dado serviço pré-existente oferece funcionalidades com propósitos afins ao serviço sendo desenvolvido, há violação da autonomia de tempo de *design*, o que é um indício de que novas operações deveriam ser agregadas aos serviços pré-existentes, e não a um novo serviço.
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Requisitos

Artefatos Produzidos:	• Contratos atualizados para posicionar operações de acordo com autonomia de tempo de *design*
Identificador	**UNR.PRAE.CONC.0020**
Nome:	Projetar ausência de estado
Descrição:	O princípio ausência de estado está relacionado à delegação de informações de estado pelo serviço a um mecanismo externo. Este princípio visa maximizar a escalabilidade de um serviço pela minimização de consumo de recursos gerenciando o estado do serviço [4]. Este princípio também se relaciona ao conceito de "idempotência". Idempotência é uma propriedade segundo a qual o resultado de uma operação é sempre o mesmo para mesmas entradas, independente do número de execuções daquela operação. Analogamente, um serviço que não retém estado deve ser idempotente, fornecendo sempre o mesmo resultado para um dado conjunto fixo de entradas. De acordo com NARAYANAN [70], a revisão de código sob demanda provida pelo *Pair Programming* e *Refactoring* são práticas que auxiliam a identificar e corrigir retenção incorreta de estado. Além disso, a carga esperada para o serviço deve ser descrita nos critérios de aceitação. **Testes automatizados** de carga serão utilizados para garantir a aderência do serviço a estes requisitos a qualquer momento, conforme descrito por BECK [16]. Ainda de acordo com BECK [16], não deve haver dependências entre os testes, os quais devem rodar num ambiente neutro dentro do processo de **integração contínua**.

	O uso de **metáforas** pode também se provar útil para discussões entre equipe e PO, visando identificar se faz sentido reter as informações de negócio englobadas pela metáfora. A seção Arquitetura Interna apresenta as atividades a serem desempenhadas para atingir os objetivos definidos por ERL [4].
Tipo de Componente:	Concreto
Obrigatório:	Sim
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Critérios de Entrada	• Requisitos fornecidos
Critérios de Saída	• Ausência de estado projetada
Artefatos Requeridos:	• Requisitos
Artefatos Produzidos:	• Definição de informações a serem retidas • Testes automatizados
Características Atendidas:	• Ausência de estado • Integração contínua • Testes automatizados
Variantes deste Componente:	Não há
Arquitetura Interna:	

Atividade	**Projetar retenção intencional de estado**
Descrição:	Apesar da definição de ERL [4] de que serviços devem, idealmente, ser idempotentes e não reter informações de estado, eventualmente pode ser necessário fazê-lo, sob pena de não cumprir um requisito funcional. Por exemplo, em serviços que implementam lógicas de segurança e apresentam alto custo de autenticação a toda invocação, pode ser inviável não reter informações indicando que o usuário já está autenticado e dispensa nova autenticação. Desta forma, a equipe deve projetar os serviços de forma que demandem retenção mínima de estado e retenham somente informações essenciais. A equipe deve também projetar testes automatizados que verifiquem que o nível de retenção de estado está dentro do planejado.
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Requisitos
Artefatos Produzidos:	• Definição de informações a serem retidas
Atividade	**Projetar testes de carga**
Descrição:	Em casos onde seja necessário que o serviço retenha informações de estado, é necessário verificar se o mecanismo de retenção prejudica o desempenho, estabilidade e disponibilidade do serviço.

	Testes de carga devem ser projetados para realizar esta verificação. Deve-se verificar em que limites o serviço consegue continuar mantendo o nível de acordo serviço e atendendo aos requisitos funcionais. Estes testes executando num ambiente de integração contínua também permitem identificar se alguma modificação introduziu perda ao desempenho do serviço.
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Requisitos
Artefatos Produzidos:	• Testes automatizados
Atividade	**Projetar testes de verificação de idempotência**
Descrição:	Quando não houver necessidade de reter informações de estado, devem ser projetados testes verificando que esta definição não é violada. Testes deste tipo devem verificar que, dado um conjunto de entradas, o resultado será sempre o mesmo, independente da quaisquer fatores como, por exemplo, número de execuções ou data corrente do sistema. Esta verificação também pode ser feita através de testes onde usuários concorrentes requerem informações personalizadas e o retorno é sempre o mesmo, comprovando que o estado de um usuário não é retido e que informações de um indivíduo não são indevidamente expostas a outro usuário.

	De forma análoga ao componente anterior, um ambiente de integração contínua permite à equipe ser notificada imediatamente quando tal violação for introduzida no código.
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Requisitos
Artefatos Produzidos:	• Testes automatizados
Identificador	**UNR.PRDE.CONC.0021**
Nome:	Projetar descoberta
Descrição:	Este princípio visa aumentar a qualidade e padronização dos metadados contidos em um contrato, de forma a facilitar sua descoberta, interpretação e uso por agentes humanos ou automáticos. Essas informações devem refletir o propósito e as capacidades do serviço, permitindo ao usuário buscar em um repositório por serviços que resolvam sua necessidade de negócio dentro de parâmetros de qualidade de serviço aceitáveis. Para atingir tal objetivo, as informações contidas nas *User Stories* e nos critérios de aceitação servem de base para publicar, no contrato, o propósito e capacidades (funcionais e não-funcionais) do serviço **(a)**.

	Além disso, o uso conjunto das práticas de **Testing** e **Continuous Integration** permite aferir que um serviço é adequadamente descoberto a qualquer momento. Ou seja, é possível criar testes automatizados que executem, com uma dada frequência, uma busca no repositório de serviços utilizando um conjunto de parâmetros a fim de verificar se o serviço desejado é retornado. Conforme definido por ERL [4], o contrato é formado por duas partes: uma técnica e outra não-técnica. O primeiro visa à descoberta e interpretação automáticas por máquinas, o segundo visa ao uso humano do contrato. Assim, ao projetar um serviço para descoberta, a equipe deve projetar seu contrato considerando não só aspectos técnicos como políticas de segurança do padrão WS-* [4], mas também deve garantir que o contrato especifica informações conforme a taxonomia da organização, está corretamente disponibilizado no UDDI [74], especifica políticas e obedece a padrões organizacionais para localização de serviço. O PO, além disso, pode contribuir fazendo testes de busca do serviço no repositório, de acordo com a utilização esperada dessas buscas, e verificando se o mesmo é de fácil descoberta. Por fim, o uso de **Small Releases** e **Refactoring** permite ao time e PO atualizar constantemente o contrato, seja para publicar meta-dados de novos serviços ou refinar meta-dados existentes.
Tipo de Componente:	Concreto
Obrigatório:	Sim

Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Critérios de Entrada	• Requisitos fornecidos
Critérios de Saída	• Descoberta projetada
Artefatos Requeridos:	• Requisitos
Artefatos Produzidos:	• Contrato atualizado
Características Atendidas:	• Descoberta
Variantes deste Componente:	Não há
Arquitetura Interna:	Não há
Identificador	**UNR.DEES.CONC.0022**
Nome:	Definir estratégia de evolução de serviço pré-existente
Descrição:	Conforme identificado por ERL [4], SOA é de natureza dinâmica, e mudanças fazem parte do seu ciclo de vida. Prioridades e regras de negócio mudam com frequência, e um dos objetivos de SOA é exatamente oferecer flexibilidade para acolher tais mudanças com agilidade. No entanto, a modificação de serviços apresenta uma complexidade maior, quando comparado a softwares ditos “tradicionais”. As partes impactadas pela mudança, com frequência, são externas às fronteiras organizacionais.

	Dependendo do nível de reuso de um serviço dentro da organização, uma modificação problemática pode acarretar grande impacto sobre a própria organização. Assim, este componente tem por objetivo definir estratégias de evolução de serviços a fim de minimizar e tratar o impacto sobre consumidores. No caso de modificação do comportamento externo ou modificação do contrato, a equipe deve identificar e negociar com consumidores de forma a estabelecer uma estratégia de comum acordo para evolução do serviço, com impacto mínimo entre as partes. No caso de evoluções disruptivas, uma possível estratégia é a publicação das mudanças, identificadas por uma nova versão, em um ambiente separado da versão corrente, utilizada pelos consumidores, permitindo aos consumidores planejar sua migração para o novo endpoint dentro da janela de tempo acordada. Nesse caso, há que se estabelecer também uma estratégia de evolução da versão anterior em caso de ocorrência de defeitos. Uma possível estratégia seria disponibilizar as correções em ambos os ambientes enquanto ambos os serviços estiverem no ar. Outra possível estratégia seria disponibilizar as correções somente na versão mais recente, forçando os consumidores a migrarem seu consumo com mais rapidez. De forma geral, a estratégia escolhida deve, preferencialmente manter a suíte de testes existente estável ao mesmo tempo em que se inserem novos testes e código de produção que atendem aos requisitos modificados.
Tipo de Componente:	Concreto

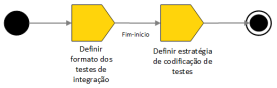
Obrigatório:	Não
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Critérios de Entrada	• Há serviços pré-existentes a serem modificados
Critérios de Saída	• Atingido consenso sobre a estratégia de evolução
Artefatos Requeridos:	• Código de produção do serviço a ser evoluído • Contrato do serviço a ser evoluído • Testes pré-existentes do serviço a ser evoluído
Artefatos Produzidos:	• Código de produção • Contrato • Testes
Características Atendidas:	• Modificação de contrato
Variantes deste Componente:	Não há
Arquitetura Interna:	Não há
Identificador	**UNR.DERE.CONC.0023**
Nome:	Definir responsáveis pelas tarefas

Descrição:	O propósito deste componente é definir que pessoas, dentro da equipe de desenvolvimento, serão responsáveis por quais tarefas durante o desenvolvimento da funcionalidade. Isto será feito de forma progressiva e dinâmica, a cada funcionalidade, sendo aberto a mudanças a qualquer momento. Complementarmente, este componente visa definir todas as tarefas a serem executadas pela equipe, ao longo do desenvolvimento, até que a funcionalidade seja considerada completa e pronta para validação pelo PO, conforme representado na seção Arquitetura Interna a seguir. Ao fazer o planejamento de tarefas e responsáveis, a equipe fica responsável por decidir o ferramental necessário à execução da tarefa, de acordo com sua configuração (co-localizada ou distribuída). Assim como nos componentes anteriores, este ferramental pode incluir ferramentas de áudio/videoconferência, salas de reunião, registro do planejamento, entre outras. De forma a permitir o acompanhamento instantâneo do desenvolvimento por todos os envolvidos, equipes distribuídas devem preferencialmente registrar seu progresso em meio digital e online. Este componente é executado ao início de cada funcionalidade planejada para a iteração. Ao final deste componente, o planejamento de tarefas de cada funcionalidade priorizada para a iteração é registrado, em conjunto com os responsáveis pelas primeiras tarefas.
Tipo de Componente:	Concreto

Obrigatório:	Sim
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Critérios de Entrada	• Testes de aceitação automatizados • Contrato mínimo identificado • Contrato atualizado com informações de baixo acoplamento • Contrato atualizado com informações de abstração • Reuso projetado • Autonomia projetada • Ausência de estado projetada • Descoberta projetada
Critérios de Saída	• Responsáveis atribuídos à tarefas
Artefatos Requeridos:	• *Backlog* da iteração • Requisitos
Artefatos Produzidos:	• Planejamento de tarefas a serem executadas até a conclusão de cada funcionalidade, com respectivos responsáveis, se aplicável
Características Atendidas:	• Equipes distribuídas
Variantes deste Componente:	Não há
Arquitetura Interna:	

Atividade	**Planejar tarefas**
Descrição:	Uma vez finalizada a *Iteration planning meeting*, a equipe deve, em conjunto, delinear as tarefas necessárias para a conclusão do desenvolvimento da funcionalidade, além de definir os responsáveis por cada tarefa **(a)**. O planejamento das tarefas a serem executadas para cada funcionalidade deve ser registrado em algum meio, físico ou digital, acessível a todos os participantes do projeto (não só à equipe de desenvolvimento). O objetivo é que todos tenham ciência do estado atual do desenvolvimento a qualquer momento.
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• *Backlog* de funcionalidades • Requisitos
Artefatos Produzidos:	• Planejamento de tarefas a serem executadas até a conclusão de cada funcionalidade.
Atividade	**Atribuir responsabilidades**
Descrição:	Num primeiro momento, cada par de pessoas se atribui a responsabilidade por uma tarefa **(a)**. Conforme o desenvolvimento progride, pares de pessoas finalizam tarefas e se atribuem novas tarefas, até o final do trabalho **(b)**.
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento

Artefatos Requeridos:	• *Backlog* de funcionalidades • Requisitos
Artefatos Produzidos:	• Planejamento de tarefas a serem executadas até a conclusão de cada funcionalidade, com respectivos responsáveis, se aplicável.
Identificador	**UNR.PLET.CONC.0024**
Nome:	Planejar estratégias de teste
Descrição:	O propósito deste componente é definir as estratégias de codificação e testes unitários a serem seguidos ao longo da construção da funcionalidade.
Tipo de Componente:	Concreto
Obrigatório:	Não
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Critérios de Entrada	• Responsáveis atribuídos à tarefas
Critérios de Saída	• Formato dos testes de integração definido • Estratégia de codificação de testes definida
Artefatos Requeridos:	• Testes de aceitação automatizados • Contrato mínimo da funcionalidade • Contrato do serviço • Definição sobre conectores, adaptadores, conversões, etc., para favorecer o reuso

	• Definição sobre medida de reuso do serviço • Definição sobre impactos a modelos canônicos • Definição de informações a serem retidas • Código de produção • Testes pré-existentes
Artefatos Produzidos:	• Definição sobre o formato dos testes de integração • Definição sobre a estratégia de codificação de testes
Características Atendidas:	• TDD
Variantes deste Componente:	Não há
Arquitetura Interna:	
Atividade	**Definir formato dos testes de integração**
Descrição:	Esta atividade se destina ao estabelecimento de um consenso entre a equipe sobre a melhor forma de conduzir os testes de integração de um serviço, no que se refere a sua integração com outros serviços. De acordo com RIBAROV *et al.* [75], a natureza dinâmica e adaptável de SOA faz com que muitas técnicas de teste ditas "tradicionais" não sejam aplicáveis ao teste de serviços **(a)**. Há diversas técnicas e formas de teste que podem ser aplicadas ao desenvolvimento de serviços de acordo com, por exemplo: características do projeto; restrições de orçamento; necessidade de resultados precisos.

	Este componente prevê o uso de três possibilidades na automação dos testes de integração de um serviço. 1- **Testes de integração por meio de simulação (*mock*) de API's**: Esta alternativa se refere a cenários onde há uma API que provê isolamento entre o provedor e o consumidor de um serviço **(a)**. Neste caso, a equipe pode optar por simular o funcionamento do serviço por meio do uso da técnica de mocking da API. Ou seja, durante a execução do teste de integração, o par de programadores define o resultado esperado de uma invocação e simula a invocação da API, atendo-se ao funcionamento esperado de sua lógica de negócio **(b)**. 2- **Testes de integração por meio de simulação de serviços**: Nesta alternativa, quando a ligação entre serviços é feita de forma dinâmica, o serviço mock continua sendo necessário, bem como uma massa de dados estável e representativa. Contudo, não será necessário modificar o endereço do endpoint dinamicamente. Neste caso, conforme sugerido por RIBAROV *et al.* [75], é necessário ter um ESB habilitado para testes (*test-enabled ESB*), onde seja possível dinamicamente apontar para o serviço mock ao invés do serviço real. 3- **Testes de integração por meio de invocações reais**: A última possibilidade é a execução de testes de integração contra o serviço real. Neste caso, se houver a necessidade de saber com precisão o retorno do serviço, será necessário chegar a algum tipo de acordo com o provedor do serviço. Ao final desta atividade, a equipe deve registrar que tipo de teste de integração será utilizado, junto com uma justificativa opcional.
Participantes:	• Equipe de desenvolvimento

Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Testes de aceitação automatizados • Contrato mínimo da funcionalidade • Contrato do serviço • Definição sobre conectores, adaptadores, conversões, etc., para favorecer o reuso • Definição sobre medida de reuso do serviço • Definição sobre impactos a modelos canônicos • Definição de informações a serem retidas • Código de produção • Testes pré-existentes
Artefatos Produzidos:	• Definição sobre o formato dos testes de integração
Atividade	**Definir estratégia de codificação de testes**
Descrição:	Este componente objetiva definir a estratégia pela qual os testes (e, por conseguinte, o código de produção) será gerado. Esta estratégia deve ser decidida em conjunto pela equipe, de acordo com o contexto do projeto em que o componente é aplicado. De forma geral, BECK [16] define que toda atividade da equipe deve ser desempenhada em pares. Sendo feita em par, uma dada atividade é automaticamente revisada ao mesmo tempo em que é produzida. A codificação, mais especificamente, além de ser feita em pares, deve ser guiada por testes de forma iterativa e incremental, seguindo os passos abaixo:

		A1. Crie um teste, baseado em um critério de aceitação; A2.Execute o teste e verifique que ele falha; A3.Projete e implemente a forma mais simples de fazer o teste passar, e que mantém os testes anteriores passando; A4.Execute o teste e verifique que ele é bem sucedido; A5.Repita os passos acima para exercitar uma variação do critério, ou para endereçar o próximo critério de aceitação; A6.Se for possível simplificar o design, faça-o. Desta forma, ao final destes passos, o código de produção será gerado como a resposta mais simples aos testes de aceitação. Além disso, desta forma os testes não são produzidos de forma "viciada". Ou seja, os testes de aceitação definem o comportamento do código de produção. Feitos de outra forma, escritos após o código de produção, os testes muitas vezes tendem a ser menos abrangentes e rigorosos, uma vez que o programador já conhece a lógica interna, casos de falha previamente tratados, etc. Escritos após o código de produção, os testes tem um propósito maior de documentar o funcionamento já implementado do que de definir o comportamento esperado. No entanto, é sabido que programação em pares e TDD (*Test-Driven Development*) são algumas das práticas do XP de menor aceitação na indústria [44]. Há diversas questões que influenciam neste ínterim, como a cultura das organizações, preferências pessoais dos profissionais, entre outras.

Assim, a equipe deve definir quais das abordagens a seguir será a escolhida para construção dos testes e do código de produção:

1- **Testes escritos em par antes do código de produção**: Esta abordagem é a recomendada por BECK [16]. Neste caso, os passos a serem desempenhados pelo par de programadores são os passos A1-A6 definidos acima. A equipe fica responsável por decidir o ferramental necessário a execução da tarefa de acordo com a sua configuração. Assim como nos componentes anteriores, este ferramental pode incluir ferramentas de áudio/videoconferência, salas de reunião, entre outras, de acordo com a localização dos membros da equipe.

2- **Testes escritos individualmente antes do código de produção**: Esta abordagem é semelhante à recomendada por BECK [16], com a diferença do número de pessoas que a executa. Apesar de eliminar a necessidade de ferramental extra para sua execução, se perde a vantagem da revisão automática dos testes e código produzidos, requerendo a execução do componente UNR.COSE.CONC.0030, para que código e testes sejam revisados por outro membro da equipe, a procura de pontos de melhoria.

3- **Testes escritos em par depois do código de produção**: Neste caso, o propósito do teste deixa de ser a automação a execução de um critério de aceitação e passa a ser a documentação do funcionamento atual daquele código.
Neste caso, os passos a serem desempenhados são ligeiramente diferentes:

P1.Escreva o código de produção que atende ao critério de aceitação.

	P2.Crie um teste que documente o código escrito; P3.Implemente o teste de forma que execute sem falha, ao mesmo tempo em que mantém os testes anteriores passando; P4.Execute o teste e verifique que ele passa; P5.Repita os passos acima para exercitar uma variação do código, ou para endereçar o próximo código; P6.Se for possível simplificar o design, faça-o. Uma vez que esta estratégia seja escolhida, o uso de um par de pessoas dispensa a execução do componente UNR.COSE.CONC.0030. 4- **Testes escritos** **individualmente depois do código de produção**: Novamente, o propósito dos testes é de documentação de uma lógica de negócio já codificada. Sendo executado individualmente, dispensa ferramental para execução por times distribuídos, mas requer revisão de código, por meio da execução do componente UNR.COSE.CONC.0030. Ao final desta atividade, a equipe deve registrar a estratégia de codificação de testes a ser utilizada, junto com uma justificativa opcional.
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Testes de aceitação automatizados • Contrato mínimo da funcionalidade • Contrato do serviço

	• Definição sobre conectores, adaptadores, conversões, etc., para favorecer o reuso • Definição sobre medida de reuso do serviço • Definição sobre impactos a modelos canônicos • Definição de informações a serem retidas • Código de produção • Testes pré-existentes
Artefatos Produzidos:	• Definição sobre a estratégia de codificação de testes
Identificador	**UNR.COTI.ABST.0025**
Nome:	Codificar guiado por testes de integração
Descrição:	Ao executar este componente, a equipe deve implementar os testes de integração do serviço, de acordo com o formato e a estratégia definidos no componente UNR.PLET.CONC.0024, e com as decisões de *design* acordadas nos componentes UNR.PRBA.CONC.0016, UNR.PRAB.CONC.0017, UNR.PRRE.CONC.0018, UNR.PRAU.CONC.0019, UNR.PRAE.CONC.0020 e UNR.PRDE.CONC.0021. Deve-se manter, ainda, a estabilidade dos testes pré-existentes. Ao final da execução deste componente, será gerado código de produção e testes aderentes ao formato e estratégia definidos.
Tipo de Componente:	Abstrato
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento

Critérios de Entrada	• Formato dos testes de integração definidos • Estratégia de codificação de testes definida • Responsáveis definidos
Critérios de Saída	• Testes de integração codificados • Código de produção codificado
Artefatos Requeridos:	• Requisitos • Planejamento de tarefas a serem executadas até a conclusão de cada funcionalidade, com respectivos responsáveis, se aplicável • Definição sobre a estratégia de codificação de testes
Artefatos Produzidos:	• Testes de integração • Código de produção
Características Atendidas:	• Testes distribuídos • TDD • Testes de integração sem barramento • Testes de integração sem invocações reais
Variantes deste Componente:	• UNR.COMA.CONC.0026 • UNR.COMS.CONC.0027 • UNR.COTR.CONC.0028

Arquitetura Interna	Não há
Identificador	**UNR.COMA.CONC.0026**
Nome:	Codificar mocks de API para teste de integração entre serviços
Descrição:	Este componente se relaciona ao formato de “Testes de integração por meio de simulação (*mock*) de API’s”, definido no componente UNR.PLET.CONC.0024, e se destina à codificação de testes de integração por meio de API simulada. Nesta variante, os passos definidos no componente UNR.DEEC.CONC.0023 devem ser acrescidos de outro, destinado a definir e simular o comportamento esperado da API. Por exemplo, na estratégia de testes escritos antes do código de produção, a definição final dos passos a serem executados pela equipe seria: A1.Crie um teste, baseado em um critério de aceitação e que avalie uma interação entre os serviços; A2.Execute o teste e verifique que ele falha; A3.Defina e simule o comportamento esperado da API quando for invocada; A4.Projete e codifique a forma mais simples de fazer o teste passar, e que mantém os testes anteriores passando; A5.Execute o teste e verifique que ele é bem sucedido;

	A6.Repita os passos acima para exercitar uma variação do critério, ou para endereçar o próximo critério de aceitação; A7.Se for possível simplificar o *design*, faça-o.
Tipo de Componente:	Concreto
Obrigatório:	Não
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Critérios de Entrada	• Formato dos testes de integração definidos • Estratégia de codificação de testes definida • Responsáveis definidos
Critérios de Saída	• Testes de integração codificados • Código de produção codificado
Artefatos Requeridos:	• Requisitos da funcionalidade a ser trabalhada • Definição sobre a estratégia de codificação de testes
Artefatos Produzidos:	• Testes de integração • Código de produção
Características Atendidas:	• Testes distribuídos • TDD • Testes de integração sem barramento

	• Testes de integração sem invocações reais
Variantes deste Componente:	Não há
Arquitetura Interna:	Não há
Identificador	**UNR.COMS.CONC.0027**
Nome:	Codificar mocks de serviços para teste de integração entre serviços
Descrição:	Este componente se relaciona ao formato de "Testes de integração por meio de simulação de serviços", declarado no componente UNR.PLET.CONC.0024, e se destina à codificação de testes de integração por meio de serviços simulados. Conforme definido anteriormente, nesta variante é necessário lançar mão de um *test-enabled ESB* ou da publicação do serviço simulado em algum ambiente, conforme definido por RIBAROV *et al.* [75], o que faz com que os passos definidos no componente UNR.PLET.CONC.0024 sejam complementados conforme a seguir: A1.Crie um teste, baseado em um critério de aceitação e que avalie uma interação entre os serviços; A2.Execute o teste e verifique que ele falha; A3.Defina o resultado esperado da operação a ser reutilizada; A4.Publique o serviço simulado (seja em um *test-enabled ESB*, seja em um ambiente a ser ligado ponto a ponto com o serviço sendo desenvolvido);

	A5.Projete e codifique a forma mais simples de fazer o teste passar, e que mantém os testes anteriores passando; A6.Execute o teste e verifique que ele é bem sucedido; A7.Repita os passos acima para exercitar uma variação do critério, ou para endereçar o próximo critério de aceitação; A8.Se for possível simplificar o design, faça-o.
Tipo de Componente:	Concreto
Obrigatório:	Não
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Critérios de Entrada	• Formato dos testes de integração definidos • Estratégia de codificação de testes definida • Responsáveis definidos
Critérios de Saída	• Testes de integração codificados • Código de produção codificado
Artefatos Requeridos:	• Requisitos da funcionalidade a ser trabalhada • Definição sobre a estratégia de codificação de testes
Artefatos Produzidos:	• Testes de integração • Código de produção

Características Atendidas:	• Testes distribuídos • TDD • Testes de integração sem invocações reais
Variantes deste Componente:	Não há
Arquitetura Interna:	Não há
Identificador	**UNR.COTR.CONC.0028**
Nome:	Codificar testes de integração entre serviços por meio de invocações reais
Descrição:	Este componente se relaciona a ao formato de "Testes de integração por meio de invocações reais", declarado no componente UNR.PLET.CONC.0024, e se destina à codificação de testes de integração que executem invocações reais de serviços legados ou de terceiros. Os passos definidos no componente UNR.PLET.CONC.0024 são complementados conforme a seguir: A1.Crie um teste, baseado em um critério de aceitação e que avalie uma interação entre os serviços; A2.Execute o teste e verifique que ele falha; A3.Verifique a massa de dados disponível no serviço legado que atende aos propósitos do teste e defina o resultado esperado do teste;

	A4.Projete e codifique a forma mais simples de fazer o teste passar, e que mantém os testes anteriores passando; A5.Execute o teste e verifique que ele é bem sucedido; A6.Repita os passos acima para exercitar uma variação do critério, ou para endereçar o próximo critério de aceitação; A7.Se for possível simplificar o design, faça-o. Note-se que, em função da estabilidade e volume dos dados disponibilizados pelo serviço legado, pode ser necessário relaxar as condições do teste de integração para considerar múltiplas possibilidades de resposta **(a)**.
Tipo de Componente:	Concreto
Obrigatório:	Não
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Critérios de Entrada	• Formato dos testes de integração definidos • Estratégia de codificação de testes definida • Responsáveis definidos
Critérios de Saída	• Testes de integração codificados • Código de produção codificado
Artefatos Requeridos:	• Requisitos da funcionalidade a ser trabalhada

	• Estratégia de codificação de testes
Artefatos Produzidos:	• Testes de integração • Código de produção
Características Atendidas:	• Testes distribuídos • TDD
Variantes deste Componente:	Não há
Arquitetura Interna:	Não há
Identificador	**UNR.COTU.CONC.0029**
Nome:	Codificar guiado por testes unitários
Descrição:	Este componente é uma das últimas partes da cadeia de codificação combinada de testes e código de produção. Conforme definido no componente UNR.AUTE.CONC.0014, a codificação se inicia com a codificação dos testes de aceitação. Estes guiarão o desenvolvimento da funcionalidade e servirão para aferir sua completude.Uma vez prontos os testes de aceitação, o código de produção é gerado em conjunto com os testes de integração e com os testes unitários. Os testes de integração visam aferir e validar o comportamento e a comunicação entre o serviço em desenvolvimento e outros serviços ou componentes externos. Os testes unitários, por sua vez, tem por finalidade fazer a mesma validação, desta vez entre as camadas internas de um serviço.

	Durante a execução deste componente a equipe deve utilizar a estratégia a definida no componente UNR.PLET.CONC.0024, respeitando as decisões de *design* acordadas nos componentes UNR.PRBA.CONC.0016, UNR.PRAB.CONC.0017, UNR.PRRE.CONC.0018, UNR.PRAU.CONC.0019, UNR.PRAE.CONC.0020 e UNR.PRDE.CONC.0021. Ao fim, será produzido o código de produção que responde da forma mais simples aos testes de aceitação. Além disso, a equipe deve garantir que os testes pré-existentes continuam executando de forma estável.
Tipo de Componente:	Concreto
Obrigatório:	Não
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Critérios de Entrada	• Estratégia de codificação de testes definida • Responsáveis definidos
Critérios de Saída	• Testes unitários codificados • Código de produção codificado
Artefatos Requeridos:	• Requisitos da funcionalidade a ser trabalhada • Estratégia de codificação de testes • Testes de aceitação
Artefatos Produzidos:	• Testes unitários

	• Código de produção
Características Atendidas:	• TDD
Variantes deste Componente:	Não há
Arquitetura Interna:	Não há
Identificador	**UNR.MOSE.COSE.0030**
Nome:	Codificar serviço
Descrição:	Este componente é opcional, só sendo obrigatória sua execução caso os componentes que tratam de testes de integração e testes unitários não tenham sido executados. Analogamente, este componente prevê que o código de produção será gerado sem os testes cobertos pelos componentes dispensados. No entanto, é de responsabilidade da equipe garantir que os testes pré-existentes continuem executando de forma estável, e que serão respeitadas as decisões de *design* acordadas nos componentes UNR.PRBA.CONC.0016, UNR.PRAB.CONC.0017, UNR.PRRE.CONC.0018, UNR.PRAU.CONC.0019, UNR.PRAE.CONC.0020 e UNR.PRDE.CONC.0021.

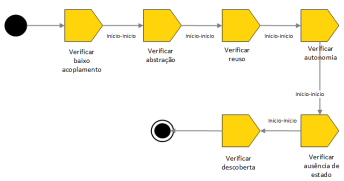
	A execução deste componente gerará código de produção aderente aos testes de aceitação, e às definições de *design*, ficando os membros da equipe responsáveis por obter os recursos necessários a sua execução (ex.: reserva de salas de reunião, ferramentas de áudio/videoconferência, *laptops*, etc) de acordo com a forma de trabalho da equipe (co-localizada ou distribuída). Após sua execução, a execução do componente UNR.RECO.CONC.0031 é obrigatória.
Tipo de Componente:	Concreto
Obrigatório:	Não
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Critérios de Entrada	• Não serão usados testes durante o desenvolvimento da funcionalidade • Responsáveis definidos
Critérios de Saída	• Código de produção escrito
Artefatos Requeridos:	• Requisitos
Artefatos Produzidos:	• Código de produção
Características Atendidas:	• Programação em pares
Variantes deste Componente:	Não há
Arquitetura Interna:	Não há

Identificador	**UNR.RECO.CONC.0031**
Nome:	Revisar código
Descrição:	Este componente está relacionado a estratégia definida pela equipe na execução do componente UNR.PLET.CONC.0024. Caso a estratégia escolhida preconize o trabalho individual em detrimento do trabalho em pares, este componente deve ser executado. Este componente também é obrigatório no caso de modificação de serviço pré-existente feita individualmente, ao invés de por pares. Ao executar este componente, um ou mais membros da equipe devem revisar os testes e código de produção produzidos, verificando a qualidade destes. De forma geral, a qualidade dos testes e do código de produção podem ser aferidos utilizando os princípios definidos por FOWLER [12] e BECK [16]. A qualidade dos testes pode ser aferida verificando, por exemplo, se: • executam com sucesso; • todos os critérios de aceitação passíveis de automação foram cobertos; • automatizam variações dos critérios que não foram inicialmente previstas; • exercitam casos limítrofes; • não há dependências entre os testes; • os testes são atemporais, ou seja, se continuarão funcionando independente da data corrente ou dos dados fornecidos por terceiros, no caso de testes de integração; • são aderentes ao formato e estratégia definidos;

	• expressam a condição que está sendo testada; • não duplicam outros testes; • têm granularidade fina, ou seja, se testam um número mínimo de condições, ao invés de testar várias condições ao mesmo tempo. A qualidade do código de produção pode ser verificada aferindo, entre outros pontos: • sua aderência aos padrões de codificação da organização; • seu respeito a boas práticas definidas pela comunidade da plataforma de desenvolvimento e pela organização; • sua legibilidade; • seu índice de complexidade ciclomática; • o grau de acoplamento entre as partes; • o grau de coesão entre as partes; • se os métodos e operações expressam seu propósito; • se não código duplicado; • se as classes tem um propósito único; • se não há dependência cíclica entre os pacotes; • se o empacotamento das classes é adequado; Se houver violações a um ou mais itens, o código deve ser atualizado imediatamente, eliminando estas violações **(a)**.
Tipo de Componente:	Concreto

Obrigatório:	Não
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Critérios de Entrada	• Estratégia de codificação de testes escolhida prevê trabalho individual
Critérios de Saída	• Testes unitários revisados • Código de produção revisado
Artefatos Requeridos:	• Testes unitários • Código de produção
Artefatos Produzidos:	• Testes unitários atualizados • Código de produção
Características Atendidas:	• Equipes distribuídas
Variantes deste Componente:	Não há
Arquitetura Interna:	Não há
Identificador	**UNR.ADPS.CONC.0032**
Nome:	Adequar princípios de SOA

Descrição:	Este componente está relacionado aos princípios de SOA definidos por ERL [4]. O objetivo deste componente é revisar os artefatos produzidos (código de produção, contrato, testes) de forma a garantir que os princípios de SOA estão sendo cumpridos dentro do nível planejado. Para tanto, diversas atividades devem ser desempenhadas, conforme a seção Arquitetura Interna a seguir. Este componente é de execução opcional quando os componentes anteriores (UNR.COTI.ABST.0025, UNR.COTU.CONC.0029, UNR.COSE.CONC.0030) tenham sido executados em pares, execução esta que provê revisão dos artefatos no próprio momento em que são produzidos. A revisão de cada artefato deve ser desempenhada por um membro da equipe diferente daquele que o produziu. Caso sejam encontradas diferenças entre o implementado e o planejado durante a execução das atividades, estas devem preferencialmente ser corrigidas imediatamente **(a)**.
Tipo de Componente:	Concreto
Obrigatório:	Sim
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Critérios de Entrada	• Serviço foi codificado de forma individual
Critérios de Saída	• Testes revisados • Código de produção revisado • Contrato revisado

Artefatos Requeridos:	• Testes • Código de produção • Contrato
Artefatos Produzidos:	• Testes • Código de produção • Contrato
Características Atendidas:	• Equipes distribuídas • Contrato • Baixo Acoplamento • Abstração • Autonomia • Ausência de Estado • Descoberta • Composabilidade • Modificação de contrato • Reuso
Variantes deste Componente:	Não há
Arquitetura Interna:	

Atividade	**Verificar baixo acoplamento**
Descrição:	Nesta atividade, a equipe deve verificar se as definições acordadas no componente UNR.PRBA.CONC.0016 estão sendo atendidas nos artefatos construídos. Essa verificação inclui averiguar se os tipos de acoplamento propostos por ERL [4] estão sendo atendidos conforme planejado: acoplamento da lógica ao contrato, do contrato a lógica, do contrato a tecnologia, do contrato a implementação e do contrato ao escopo funcional. Inclui também verificar o código de produção utiliza o tipo de conexões físicas e o estilo de comunicação planejados, além da adequação das estruturas de dados, tipagem e padrão de interação.
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Código de produção • Contrato • Testes
Artefatos Produzidos:	• Código de produção • Contrato • Testes
Atividade	**Verificar reuso**

Descrição:	Ao executar esta atividade a equipe deve revisitar as decisões tomadas na execução do componente UNR.PRRE.CONC.0018, e verificar se os artefatos produzidos atendem a medida de reuso determinada, se os modelos canônicos foram modificados e disponibilizados a outros consumidores em tempo hábil, se os consumidores foram notificados sobre uma nova versão do serviço.
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Código de produção • Contrato • Testes
Artefatos Produzidos:	• Código de produção • Contrato • Testes
Atividade	**Verificar autonomia**
Descrição:	Nesta atividade, que se relaciona ao componente UNR.PRAU.CONC.0019, a equipe deve verificar se os testes planejados foram implementados, além de averiguar se as fronteiras funcionais projetadas estão sendo respeitadas.
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Código de produção

	• Contrato • Testes
Artefatos Produzidos:	• Código de produção • Contrato • Testes
Atividade	**Verificar ausência de estado**
Descrição:	Durante esta atividade, a equipe deve verificar se as definições acordadas no componente UNR.PRAE.CONC.0020 estão sendo atendidas nos artefatos construídos. Isto inclui verificar se a retenção de estado está dentro do planejado e se a carga planejada está sendo suportada.
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Código de produção • Contrato • Testes
Artefatos Produzidos:	• Código de produção • Contrato • Testes
Atividade	**Verificar descoberta**

Descrição:	Durante esta atividade, a equipe deve verificar se a documentação atende aos padrões da organização, conforme definições do componente UNR.PRDE.CONC.0021.
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Código de produção • Contrato • Testes
Artefatos Produzidos:	• Código de produção • Contrato • Testes
Identificador	**UNR.DISE.ABST.0033**
Nome:	Disponibilizar serviço
Descrição:	Ao final do desenvolvimento da funcionalidade, o serviço deve ser disponibilizado, seja em preparação para a demonstração do produto da iteração, seja como parte de uma estratégia de disponibilização de modificações a um consumidor.

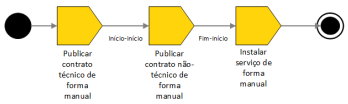
	Conforme discutido nos princípios de **Contrato Padronizado** e **Descoberta de Serviço**, a forma e o conteúdo do contrato do serviço **(a)** têm relação direta com sua facilidade de descoberta e reuso. Além disso, conforme ERL [4], o contrato de um serviço é formado por duas partes: uma técnica, direcionada à interpretação por máquinas; outra não-técnica, visando prover informações textuais a humanos. Assim, ao final da execução deste componente, além do serviço, devem também ser disponibilizadas as duas partes de seu contrato, ambas aderentes aos padrões estabelecidos durante o *Design* do serviço. Assim, neste componente, os membros da equipe de desenvolvimento, trabalhando em par, são responsáveis por diversas tarefas, conforme descrito na seção Arquitetura Interna a seguir. Caso seja identificada alguma falha no contrato, esta deve ser corrigida imediatamente, de modo a não prejudicar o compromisso estabelecido com o PO no início da Sprint, bem como o andamento da mesma.
Tipo de Componente:	Abstrato
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Critérios de Entrada	• Produto da iteração está pronto para validação
Critérios de Saída	• Serviço disponível para validação • Contratos técnico e não-técnico do serviço publicados

Artefatos Requeridos:	• Código de produção
Artefatos Produzidos:	Não há
Características Atendidas:	• Integração contínua • Disponibilização de serviço de forma manual
Variantes deste Componente:	• UNR.DIAU.CONC.0034 • UNR.DIMA.CONC.0035
Arquitetura Interna	Não há
Identificador	**UNR.DIAU.CONC.0034**
Nome:	Disponibilizar serviço por meio automático
Descrição:	Nesta variante, todas as atividades são executadas de modo automático, integrado ao ciclo de integração contínua. A ferramenta usada para integração contínua deve ser configurada para, ao final de uma construção bem sucedida (onde não há erros de compilação e todos os testes passam), fazer a instalação do serviço no(s) ambiente(s) apropriado(s).
Tipo de Componente:	Concreto
Obrigatório:	Sim
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento

Critérios de Entrada	• Produto da iteração está pronto para validação
Critérios de Saída	• Produto da iteração publicado para validação
Artefatos Requeridos:	• Código de produção • Contrato
Artefatos Produzidos:	• Código de produção • Contrato
Características Atendidas:	• Integração contínua
Variantes deste Componente:	Não há
Arquitetura Interna:	
Atividade	**Publicar contrato técnico de forma automática**
Descrição:	Antes da disponibilização do serviço para demonstração, os membros devem publicar o contrato técnico, o qual basicamente será composto de informações como, por exemplo, a definição do WSDL, definições de *schemas* em XSD ou políticas definidas segundo o padrão WS-Policy, no caso de serviços SOAP. No caso de serviços RESTful, o contrato técnico pode ser composto pelo documento WADL (*Web Application Description Language*). Tipicamente, essas informações são automaticamente disponibilizadas quando o serviço é instalado no servidor de aplicação, bastando registrar onde estas estarão disponíveis durante a demonstração do produto.

	Caso esta automação não seja possível, a equipe de desenvolvimento fica responsável por automatizar a publicação do contrato. Isto pode ser feito, por exemplo, por meio de scripts executados durante o ciclo de integração contínua do projeto.
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Código de produção • Contrato
Artefatos Produzidos:	• Não há
Atividade	**Publicar contrato não-técnico de forma automática**
Descrição:	O contrato não-técnico fornece informações importantes, como, por exemplo, as exemplificadas por ERL [4]: número de usuários concorrentes que o serviço foi projetado para suportar; informações sobre períodos programados para indisponibilidade; limitações de tamanho de mensagem; ou informações necessárias para busca de serviços, como, por exemplo, sua categorização dentro da taxonomia da empresa. A equipe de desenvolvimento deve disponibilizá-las de forma automática em local acordado com o PO, para uso durante a demonstração do produto. Novamente, esta automação pode ser feita por meio de scripts executados durante o ciclo de integração contínua do projeto.
Participantes:	• Equipe de desenvolvimento

Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Contrato
Artefatos Produzidos:	Não há
Atividade	**Instalar serviço de forma automática**
Descrição:	Ao final da execução do componente, o serviço deve ser instalado e disponibilizado de forma automática para uso no ambiente de demonstração do produto.
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Código de produção
Artefatos Produzidos:	• Não há
Identificador	**UNR.DIMA.CONC.0035**
Nome:	Disponibilizar serviço manualmente
Descrição:	Nesta variante, todas as atividades são executadas de forma manual. Uma ou mais pessoas do time de desenvolvimento ficam responsáveis por obter o produto da iteração e instalá-lo no(s) ambiente(s) apropriado(s).
Tipo de Componente:	Concreto
Obrigatório:	Sim
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento

Critérios de Entrada	• Produto da iteração está pronto para validação
Critérios de Saída	• Produto da iteração publicado para validação
Artefatos Requeridos:	• Código de produção • Contrato
Artefatos Produzidos:	• Código de produção • Contrato
Características Atendidas:	• Integração contínua • Disponibilização de serviço de forma manual
Variantes deste Componente:	Não há
Arquitetura Interna:	
Atividade	**Publicar contrato técnico de forma manual**
Descrição:	De forma análoga à atividade **Publicar contrato técnico de forma automática**, que consta na arquitetura interna do componente UNR.DIAU.CONC.0034, a equipe de desenvolvimento deve manualmente publicar o contrato técnico e, a seguir, registrar onde este estará disponível durante a demonstração do produto.
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Código de produção

	• Contrato
Artefatos Produzidos:	• Não há
Atividade	**Publicar contrato não-técnico de forma manual**
Descrição:	De forma análoga à atividade **Publicar contrato não-técnico de forma automática**, que consta na arquitetura interna do componente UNR.DIAU.CONC.0034, na execução deste componente a equipe de desenvolvimento fica responsável por manualmente disponibilizar o contrato não-técnico, em local acordado com o PO, para uso durante a demonstração do produto.
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Contrato
Artefatos Produzidos:	Não há
Atividade	**Instalar serviço de forma manual**
Descrição:	De forma análoga à atividade **Instalar serviço de forma automática**, que consta na arquitetura interna do componente UNR.DIAU.CONC.0034, a equipe de desenvolvimento deve instalar o serviço e disponibilizá-lo para uso no ambiente de demonstração do produto.
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Artefatos Requeridos:	• Código de produção

Artefatos Produzidos:	• Não há
Identificador	**UNR.EXTI.CONC.0036**
Nome:	Executar testes integrados manuais
Descrição:	Este componente visa complementar os testes automatizados, no sentido de que a equipe deve executar testes manuais de forma a verificar variações dos critérios de aceitação, casos limítrofes, integração com demais sistemas e serviços, além de executar testes de aceitação que não tenham sido possíveis de automatizar, aferindo, assim, o funcionamento integrado da solução como um todo. De forma geral, este componente se concentra nos requisitos funcionais e não-funcionais previstos pelo PO, e suas variações. Caso sejam encontrados defeitos, estes devem ser registrados para tratamento imediato pela equipe, sob pena da funcionalidade não poder se entregue ao PO, prejudicando o andamento do projeto.
Tipo de Componente:	Concreto
Obrigatório:	Sim
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Critérios de Entrada	• Produto da iteração publicado para validação
Critérios de Saída	• Testes manuais executados
Artefatos Requeridos:	• Produto da iteração

	• Requisitos
Artefatos Produzidos:	• Defeitos
Características Atendidas:	• Testes distribuídos
Variantes deste Componente:	Não há
Arquitetura Interna:	Não há
Identificador	**UNR.EXTE.CONC.0037**
Nome:	Executar testes exploratórios
Descrição:	Este componente tem por objetivo testar a funcionalidade em relação a situações imprevistas como, por exemplo, mal uso por consumidores bem ou mal intencionados ou respostas a situações não cobertas nos requisitos. A equipe deve testar variações que não tenham sido previstas nos critérios de aceitação e/ou que não sejam passíveis de automatização. Caso sejam encontrados defeitos, estes devem ser registrados e tratados imediatamente pela equipe, sob pena da funcionalidade não poder se entregue ao PO, prejudicando o andamento do projeto.
Tipo de Componente:	Concreto
Obrigatório:	Sim
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento

Critérios de Entrada	• Produto da iteração publicado para validação
Critérios de Saída	• Testes exploratórios executados
Artefatos Requeridos:	• Produto da iteração
Artefatos Produzidos:	• Defeitos
Características Atendidas:	• Testes distribuídos
Variantes deste Componente:	Não há
Arquitetura Interna:	Não há
Identificador	**UNR.DEPI.CONC.0038**
Nome:	Demonstrar produto da iteração
Descrição:	Este componente tem por objetivo apresentar ao PO o funcionamento das funcionalidades trabalhadas ao longo da iteração. Este componente é executado na reunião de demonstração da iteração (*Iteration planning meeting*), onde a equipe fica responsável por demonstrar que o produto atende aos critérios de aceitação expressos pelo PO e ao conceito de "Pronto" previamente acordado.

	Isso pode ser feito de diversas formas: a equipe pode executar os critérios de aceitação iniciais (descritos em formato textual pelo PO no início da iteração) ao longo da reunião, a equipe pode demonstrar o resultado de execução da suíte de testes e/ou o próprio PO pode testar o funcionamento dos serviços a partir dos testes de aceitação ou variações destes. Ao longo da demonstração, se forem encontrados defeitos ou identificadas melhorias, estes devem ser catalogados para priorização pelo PO. Ao final da reunião de demonstração da iteração, este componente também se finaliza. O PO fica responsável por registrar a aceitação do produto e/ou definir se os defeitos e melhorias serão priorizados para a próxima iteração ou não. Por exemplo, se os defeitos encontrados forem de baixo impacto e não impedirem o uso do serviço, o PO pode optar por deixar estes defeitos em segundo plano e dar tratamento a funcionalidades críticas de negócio que tragam maior retorno sobre o investimento. Uma vez que funcionalidades mais críticas tenham sido entregues, o PO pode optar por priorizar novamente defeitos de baixo impacto. Caso a equipe de desenvolvimento não pertença a mesma organização do PO ou do cliente, no caso de defeitos encontrados durante a demonstração da iteração, o custo de correção deve ser arcado pela equipe de desenvolvimento, sem ônus ao cliente. No caso de melhorias, o custo é do PO / cliente.

	Caso o produto da iteração seja rejeitado completamente pelo PO por algum motivo (ex.: o produto entregue foge completamente as necessidades de negócio ou não apresenta desempenho aceitável), o PO deverá registrar o motivo da rejeição, bem como as diferenças em relação ao especificado. Neste caso, PO e equipe deverão negociar a ação adequada, como, por exemplo, mudanças na priorização de funcionalidades do *Backlog* para acomodar a reconstrução completa do produto, um novo ciclo de levantamento de requisitos, a reescrita de requisitos ou, em casos mais extremos, o cancelamento do projeto.
Tipo de Componente:	Concreto
Obrigatório:	Sim
Participantes:	• Equipe de desenvolvimento
Responsáveis	• PO
Critérios de Entrada	• Produto da iteração disponível para demonstração
Critérios de Saída	• Produto aceito pelo PO • *Backlog* atualizado com nova priorização
Artefatos Requeridos:	• Produto da iteração
Artefatos Produzidos:	• *Backlog* atualizado (em caso de repriorização); • Notificação do PO de que o produto da iteração foi aceito (totalmente ou com defeitos); ou

	• Notificação do PO de que o produto da iteração não foi aceito;
Características Atendidas:	• Iteration demonstration
Variantes deste Componente:	Não há
Arquitetura Interna:	Não há
Identificador	**UNR.REMP.CONC.0039**
Nome:	Realizar Mega Planning
Descrição:	Feita após o planejamento de iteração de um time A, seu objetivo é juntar um membro de cada equipe e sincronizar o que está planejado para a próxima Sprint do time A, identificando conflitos e pontos em comum entre histórias de times trabalhando em "temas" distintos. Segundo MARANZATO, NEUBERT & HERCULANO [20] esta sincronização é fundamental, pois permite aos times acompanhar o andamento global do produto e identificar soluções prontas para problemas comuns.
Tipo de Componente:	Concreto
Obrigatório:	Sim
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Critérios de Entrada	• Planejamento de iteração de algum time realizado
Critérios de Saída	• Reunião realizada

Artefatos Requeridos:	• *Backlog* planejado para a iteração de cada time
Artefatos Produzidos:	• Identificação de conflitos e pontos em comum entre histórias de times trabalhando em temas distintos
Características Atendidas:	• Equipes distribuídas
Variantes deste Componente:	Não há
Arquitetura Interna:	Não há
Identificador	**UNR.RESM.CONC.0040**
Nome:	Realizar Stand-Up Meeting
Descrição:	A equipe deve realizar reuniões diárias de 10 a 20 minutos de duração, em horário acordado internamente por cada equipe, com o objetivo de permitir a todos os membros da equipe ter ciência do andamento das tarefas de cada membro da equipe, identificar impedimentos ao andamento da iteração e atribuir responsáveis a resolução destes.
Tipo de Componente:	Concreto
Obrigatório:	Sim
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Critérios de Entrada	• É hora de realizar a reunião
Critérios de Saída	• Reunião realizada

Artefatos Requeridos:	• Andamento das funcionalidades
Artefatos Produzidos:	• Impedimentos identificados, com respectivos responsáveis
Características Atendidas:	• Equipes distribuídas
Variantes deste Componente:	Não há
Arquitetura Interna:	Não há
Identificador	**UNR.REMS.CONC.0041**
Nome:	Realizar Mega Stand-Up
Descrição:	As equipes que colaboram em um projeto devem realizar uma reunião de 10 a 20 minutos, no meio da iteração, onde todos os integrantes de todas as equipes trabalhando naquele release discutem se a iteração está dentro do planejado, e os impactos a serem esperados, em caso negativo. O objetivo é sincronizar os times e atribuir responsáveis para resolver imediatamente os impedimentos.
Tipo de Componente:	Concreto
Obrigatório:	Sim
Participantes:	• Equipe de desenvolvimento
Responsáveis	• Equipe de desenvolvimento
Critérios de Entrada	• É hora de realizar a reunião
Critérios de Saída	• Reunião realizada

Artefatos Requeridos:	• Andamento das funcionalidades de cada equipe
Artefatos Produzidos:	• Impedimentos identificados, com respectivos responsáveis
Características Atendidas:	• Equipes distribuídas
Variantes deste Componente:	Não há
Arquitetura Interna:	Não há

8. Apêndice C - Considerações sobre a LPS

Este apêndice apresenta considerações e boas práticas relacionadas a a Linha de Processo de Software para Soluções Orientadas a Serviço proposta nesta dissertação.

Componente / atividade a que se refere:	Avaliar requisitos
Considerações:	No que se refere ao registro de pontos a serem esclarecidos junto ao PO **(a)**, num caso onde, por exemplo, os requisitos sejam armazenados como documentos em uma ferramenta de gerência de configuração, a equipe e o PO podem chegar ao acordo de registrar os pontos a serem esclarecidos como anotações no documento. Já no caso de empregar uma ferramenta wiki, pode-se chegar ao acordo de se registrá-los na própria página wiki que contém os requisitos. No que se refere ao prazo máximo de

	avaliação de requisitos **(b)**, este prazo deve levar em consideração a capacidade do PO para produzir esclarecimentos. Por exemplo, se o PO não consegue produzir respostas em menos de 3 dias, o prazo máximo não pode se estender até o dia anterior à próxima reunião de planejamento de iteração, pois o PO não terá tempo hábil para produzir respostas. A antecedência com que os requisitos devem ser disponibilizados à equipe deve ser acordada entre PO e equipe, baseando-se na disponibilidade e capacidade de produzir respostas de ambos os lados. Por exemplo, se o PO tem capacidade de esclarecer os requisitos com rapidez, mas tem poucos horários livres, recomenda-se que os requisitos sejam disponibilizados com antecedência suficiente para que o PO possa encontrar espaço em sua agenda para elucidá-los antes da reunião de planejamento da iteração. Analogamente, se a equipe consegue validar as respostas do PO com

	rapidez, mas tem dificuldades em encontrar o momento ideal para mobilizar uma ou mais pessoas para verificar estas respostas, então isto deve ser levado em consideração ao estabelecer o acordo de antecedência de disponibilização dos requisitos. Recomenda-se, também, que, a qualquer N-ésima execução deste componente, seja feita a verificação de um número de requisitos maior do que a velocidade estimada da equipe. Desta forma, caso haja muitas indefinições e pontos a esclarecer em um dado requisito priorizado para a iteração N+1, este pode ser substituído por uma ou mais funcionalidades cujos requisitos já estão maduros, enquanto o PO trabalha no esclarecimento dos requisitos problemáticos. Dessa forma, evita-se que a equipe fique parada, sem ter requisitos nos quais trabalhar.
Componente / atividade a que	Avaliar tempo necessário para esclarecer requisitos

se refere:	
Considerações:	O tempo para produzir esclarecimentos **(a)** deve levar em consideração não só a agenda do PO e sua velocidade para produzir esclarecimentos, mas também o tempo disponível até o início da próxima iteração, e a velocidade com que a equipe consegue verificar se as dúvidas foram de fato esclarecidas. Se o PO só consegue produzir os esclarecimentos no dia anterior ao início da próxima *Sprint*, dificilmente a equipe terá tempo de validar os esclarecimentos a tempo.
Componente / atividade a que se refere:	Sugerir desmembramento de *User Stories*
Considerações:	Tipicamente, uma *User Story* candidata a desmembramento é uma que visa atender um número elevado de propósitos ao mesmo tempo. Por exemplo, considere um sistema de gerenciamento de rentabilidade de anúncios publicitários, onde uma *User Story* é denominada “gerenciar retorno gerado aos clientes”. O termo

	"gerenciar" usualmente engloba uma gama diversa de atividades, como, por exemplo, verificar a taxa de conversão de novos clientes via *banners*, verificar a relação entre o custo investido em material publicitário e a quantidade de novos clientes que citam esse material como motivo de sua adesão, entre outras atividades. Neste caso, idealmente a *User Story* original "gerenciar retorno..." deveria ser desmembrada em diversas histórias como "extrair taxa de conversão via *banner*", "extrair custo-benefício de material publicitário".
Componente / atividade a que se refere:	Sugerir agrupamento de *User Stories*
Considerações:	Tipicamente, *User Stories* relativas a *bugs* ou correções rápidas (ex.: trocar um texto descritivo retornado pelo sistema) tem granularidade tão fina que são de difícil estimativa, sendo mais proveitoso agrupá-las em uma só *User*

	Story.
Componente / atividade a que se refere:	Fornecer informações de potencial de reuso
Considerações:	As informações de reuso podem estar registradas em qualquer formato acordado entre PO e equipe. Essas informações podem ser, por exemplo: • documentação de consumidores e suas necessidades específicas (funcionais e não-funcionais) sobre a funcionalidade sendo desenvolvida; • documentos indicando expectativas de reuso da funcionalidade, sem, no entanto, explicitar necessidades concretas ou necessidades de flexibilização; • documentação de serviços que já contemplam requisitos legais, funcionais ou não-funcionais inerentes a funcionalidade trabalhada (ex.: serviços de auditoria, log, padronização de cálculos).
Componente / atividade a que	Fornecer requisitos da iteração à equipe

se refere:	
Considerações:	É importante ressaltar que definição de PO é diferente da definição de cliente. Um PO é uma pessoa com conhecimento dos propósitos de negócio do sistema e com delegação suficiente do cliente para tomadas de decisão. É responsabilidade do PO fornecer informações e subsídios para que a *Sprint* prossiga sem maiores percalços. Assim, no que se refere a informações de potencial de reuso, o PO é responsável por conhecer e fornecer à equipe de desenvolvimento informações sobre serviços pré-existentes para fins de reuso. Considere-se o cenário de fábricas de *software*. Os profissionais da fábrica de *software* normalmente não tem acesso nem conhecimento suficiente da estrutura do seu cliente pra procurar serviços pré-existentes, o escopo de sua atuação em geral se limita a traduzir os requisitos recebidos em código de produção, usando as boas práticas, *frameworks* e demais artefatos indicados pela organização cliente. Se a

	organização que demanda o projeto de desenvolvimento o cliente não indicar nenhum serviço pré-existente, a fábrica de *software* não terá meios de conhecer sua existência
Componente / atividade a que se refere:	Esclarecer requisitos
Considerações:	No caso do PO optar pelo esclarecimento imediato dos requisitos, algum participante da reunião, seja o PO ou um membro da equipe de desenvolvimento, fica responsável por registrar o esclarecimento nos requisitos e notificar os demais do registro. O desenvolvimento da funcionalidade é guiado pelos critérios de aceitação, logo, caso este registro não seja feito, o trabalho não deverá ser feito, e a parte responsável pelo não registro será também responsável pelo impacto sobre o andamento do projeto. Nesse sentido, os critérios de aceitação servem também como “contrato” entre as partes, estabelecendo com clareza o trabalho a

	ser feito para atender às necessidades de negócio do cliente.
Componente / atividade a que se refere:	Repriorizar funcionalidades
Considerações:	Caso o PO opte pela repriorização de funcionalidades, fica consequentemente responsável por atualizar os requisitos com o esclarecimento, juntamente com o *Backlog*, e apresenta-lo novamente no início da iteração para a qual a funcionalidade for movida. Esta repriorização deve levar em consideração também a velocidade estimada da equipe. Por exemplo, se a equipe produz aproximadamente 20 pontos por iteração, e 15 pontos já foram planejados para uma iteração N, designar uma *Story* de 8 pontos para esta iteração ultrapassará o que a equipe de desenvolvimento consegue entregar. Assim, se for fundamental que esta funcionalidade de 8 pontos seja entregue na iteração N, será necessário modificar as prioridades das demais funcionalidades, de modo a ficar dentro

	da produtividade estimada da equipe.
Componente / atividade a que se refere:	Refinar estimativas
Considerações:	Uma vez que todos os pontos estejam esclarecidos e os requisitos sejam aceitos pela equipe de desenvolvimento, esta fica responsável por refinar suas estimativas, indicando ao PO mudanças ocorridas na pontuação das funcionalidades previamente planejadas e a pontuação das novas funcionalidades, permitindo ao PO repriorizar o *Backlog* de funcionalidades, de acordo com as prioridades de negócio daquele momento e a velocidade estimada da equipe.
Componente / atividade a que se refere:	Identificar metáforas
Considerações:	Não há uma "receita" para produzir boas metáforas num dado contexto **(a)**. KEELING & VELICHANSKY [68] oferecem sugestões de como formular uma boa metáfora, e a maioria está diretamente relacionada a

	particularidades de um projeto, ou situações vividas por um time. Por exemplo, segundo estes autores, uma boa metáfora: • Serve de guia para decisões de projeto; • Esclarece particularidades do sistema; • Se baseia em experiências compartilhadas; • Precisa de explicação para seu entendimento. Tome como exemplo um cenário fictício, onde um time está trabalhando na criação de uma composição de serviços, onde um serviço serve como ponto de entrada único da composição e um modelo canônico de objetos comuns a todos os serviços foi estabelecido. Suponha-se também que este time é formado por desenvolvedores com conhecimento prévio do padrão MVC (*Model-View-Controller*), típico de aplicações web. Nesse contexto, uma Metáfora que classifique os objetos

	canônicos sob o estereótipo *Model*, e o serviço que orquestra a composição com o estereótipo *Controller* pode ser uma boa metáfora. Esta metáfora direciona a arquitetura da composição, esclarece as responsabilidades dos serviços e o uso dos objetos canônicos. Num outro contexto, onde os desenvolvedores não conheçam o padrão MVC, esta metáfora provavelmente será ineficaz. Conforme sugerem KEELING & VELICHANSKY [68], ao iniciar o trabalho sobre um dado serviço / funcionalidade, uma Metáfora pode ser criada para estabelecer uma linguagem ubíqua, favorecendo a fluidez da comunicação com o PO e com o cliente. Esta Metáfora servirá de guia tanto para decisões arquiteturais quanto para decisões de mais baixo nível de implementação.
Componente / atividade a que se refere:	Automatizar testes de aceitação
Considerações:	Os critérios definidos pelo PO durante o

	planejamento da iteração são transformados em testes automatizados. Estes testes são executados sempre que o código da equipe é integrado em ambiente neutro, de forma a garantir, a todo momento, que todas as partes do produto sendo construído estão aderentes aos requisitos de negócio e continuam estáveis após uma dada modificação [16]. Dada a sua natureza distribuída e as questões de escalabilidade e robustez inerentes a serviços e composições **(a)**, falhas de arquitetura ou design podem trazer grande impacto para os consumidores de um serviço e para a organização. Isto aumenta a importância de se tratar tais requisitos não-funcionais nos testes de aceitação. Há diversas ferramentas e sintaxes para transformar critérios de aceitação em testes automatizados **(b)**, usando, por exemplo, o formato BDD (*Behavior-Driven Development*) adotado por ferramentas como Cucumber [1] e

[1] http://cukes.info/

	JBehave [2]. O uso de *Simple Design* **(c)** guiará a definição do contrato, os limites de responsabilidade entre os serviços, entre outras características. Estas definições devem seguir padrões pré-acordados pela equipe de forma a ser aderente aos princípios de *Collective Ownership* e *Coding Standards* do XP. A definição de simplicidade de passa pelos aspectos de comunicação, duplicação de código, quantidade de classes e métodos. O primeiro deve ser maximizado, isto é, o código deve comunicar de forma clara a qualquer programador seu propósito. Os demais devem ser minimizados [16].
Componente / atividade a que se refere:	Projetar contrato mínimo
Considerações:	Um ponto muitas vezes levantado ao se discutir a combinação SOA + XP é a questão da flexibilidade necessária aos contratos, para possibilitar reuso e diminuir *time-to-market*. De fato, há casos em que faz sentido

[2] http://jbehave.org/

	estabelecer interfaces que extrapolem a necessidade imediata. Se há uma dada condição futura que influencia uma interface priorizada no momento corrente, e se é sabido de antemão que esta condição futura é imutável, então a interface sendo trabalhada já poderia considerar esta condição [16]. Por outro lado, o trabalho priorizado não deve ser influenciado por incertezas. Por exemplo, se uma funcionalidade está sendo construída na Sprint N, e uma dada condição que pode vir a requerer mudanças no *design* não está claramente definida e/ou o time não tem condições de avaliar sua complexidade, então o *design* deve considerar a complexidade atual. Quando a condição N for priorizada e entendida apropriadamente, o *design* será evoluído, evitando que um design complexo seja criado antecipadamente para, no futuro, acabar sendo mais do que o necessário.

No que se refere à representação inicial do contrato **(a)** , ao início do trabalho sobre a primeira funcionalidade, será definida a representação inicial dos dados e políticas que atendem às necessidades daquele momento. Na iteração seguinte, se o entendimento das necessidades de negócio se modificar, o contrato deverá ser refatorado para atender às novas necessidades.

Para melhor entendimento sobre o que se referem os termos “dados” e “políticas”, considere o seguinte cenário de exemplo: a empresa fictícia Sá & Moreyra é uma manufatura de plásticos, que tem uma base de clientes fiéis em sua cidade, produzindo diversos artigos de plástico para propósitos diversos: de cadeiras a serem revendidas por atacadistas a material médico vendido diretamente a profissionais de saúde.

A Sá & Moreyra decidiu expandir seus negócios e oferecer produtos

	plásticos de seu portfólio por meio da exposição de seus produtos em motores de busca e comparação de preços. A empresa não tem por hábito participar de desenvolvimento de sistemas ou serviços e não consegue apresentar de forma precisa informações funcionais ou não-funcionais, como, por exemplo, que dados dos produtos serão expostos ou qual a carga de requisições concorrentes deve ser suportada. Esta será sua primeira empreitada no nicho de negócios diretos por meio de TI. Nesse cenário de exemplo, a funcionalidade mais prioritária é descrita pela *User Story* ”disponibilizar produtos para consulta pelo motor de busca A”. Considere também que o motor de busca A foi escolhido por ser o mais popular no nicho de mercado de venda de artigos de plástico manufaturados. Assim, dada a alta incerteza inerente ao cenário que se apresenta, os

detalhes funcionais e não-funcionais definidos terão, por definição, alta volatilidade.

Num momento inicial, em que se busca a exposição do serviço a somente um motor de busca, a representação dos objetos de entrada e saída no contrato será relativamente simples, uma vez que só há bem definidas as necessidades de um motor de busca.

Porém, dado que o plano é expor este serviço a outros motores de comparação de preços, é inevitável que o contrato mude no futuro próximo.

Assim, dado que cada motor de busca que consumirá o serviço terá uma forma de interação diferente, tentar determinar um contrato único e imutável, onde seus objetos de entrada e saída atendam a todo e qualquer motor de busca é um esforço inócuo.

Para tentar deixar o exemplo mais palpável, considere que o motor de busca A requeira que o serviço da Sá & Moreyra exponha uma operação

de nome *efetuarBuscaParaMotorA* que aceite como entrada um objeto no seguinte formato:

```
class EntradaA {
  String nome_parcial;
  double valor_minimo;
  double valor_maximo;
}
```

Considere também que o motor de busca A requeira que a saída do serviço da Sá & Moreyra esteja no seguinte formato:

```
class SaidaA {
  String código_de_retorno;
  Array nomes_dos_produtos;
  Array preços_de_cada_produto;
}
```

Num momento inicial, onde somente o motor de busca A é atendido, a definição do contrato é simples e direta. Não é necessário conter nada além de uma operação, representada conforme esperado pelo motor de busca A:

ServicoSaMoreyra

	+ efetuarBuscaParaMotorA(*EntradaA): SaidaA*	
	Não há necessidade, nesse momento inicial, que o contrato tenha uma representação diferente desta. Por mais que o planejamento seja expor o serviço a diferentes motores de busca, as necessidades dos outros motores são desconhecidas neste momento, o que tornam desnecessárias preocupações quanto a flexibilizar o contrato a ponto de suportar outros motores desconhecidos. Esta é a proposta de KROGDAHL, LUEF, & STEINDL [10], que vai de encontro aos princípios do XP, ou seja, que se deixe os detalhes do contrato ajustados às necessidades conhecidas. Conforme novas necessidades se apresentem, o contrato é ajustado de forma iterativa e incremental à nova necessidade.	
	Considere a alternativa de se tentar antecipar as necessidades de	

qualquer motor de busca, antes
mesmo de conhecer as necessidades do
motor de busca A. Neste caso, seria
necessário investir tempo na
definição de um contrato com N
operações e estruturas de dados,
podendo dar origem a um contrato
como o abaixo:

ServicoSaMoreyra
+ efetuarBuscaPorFaixaDeValores(
Number valor_minimo,
Number valor_maximo):
ListaDeProdutos
+ efetuarBuscaPorNome(
dString nome_do_produto):
ListaDeProdutos

Nesse caso, apesar do contrato
criado inicialmente ser suficientemente
flexível para oferecer dois tipos
de busca, ao se deparar com as restrições
do motor A, o trabalho seria
jogado fora, uma vez que o motor A requer
uma funcionalidade de busca
que combine faixa de valores e nome
parcial.

Os mesmos conceitos se aplicam às políticas. Por exemplo, considere que se tome a decisão de definir no contrato a seguinte política de segurança, sem conhecer previamente as necessidades ou restrições de um consumidor:

<wsp:Policy xmlns:wsp="..."
xmlns:wsse="...»
<wsse:SecurityToken
wsp:Usage="wsp:Required»
<wsse:TokenType>wsse:
Kerberosv5ST</wsse:TokenType>
</wsse:SecurityToken>
<wsse:Integrity wsp:
Usage="wsp:Required»
<wsse:Algorithm
Type="wsse:AlgSignature"
URI="w3c...aes"/>
</wsse:Integrity>
</wsp:Policy>

A política definida acima especifica que é obrigatório apresentar um *token* de segurança no padrão Kerberos, e que este deve usar o algoritmo AES.

No entanto, considere o cenário onde o motor de busca A usa o algoritmo Sha512 com todos os seus clientes atuais. Uma decisão terá de ser tomada: deixar de fazer negócios com o motor de busca mais popular do mercado, ou jogar fora o trabalho e modificar a política de segurança previamente definida?

Nem sempre uma decisão não-funcional de contrato será de tão simples resolução quanto modificar uma linha de XML. Se a seção do contrato não-técnico que define a carga de requisições concorrentes suportadas pelo serviço for definida antes de se conhecer as necessidades de um consumidor, mudanças posteriores nessa política podem demandar investimentos diversos. Por exemplo, se um contrato é definido dando suporte a 300 requisições concorrentes por segundo sem conhecer as necessidades do motor A, e este apresenta uma necessidade de suporte

	a 1000 requisições concorrentes por segundo, a mudança na definição do contrato implica em um curto prazo para obtenção de verbas para investimento em hardware de processamento paralelo, por exemplo.
	No que se refere à refatoração de contratos (b), há diversos níveis de refatoração, os quais podem gerar conflitos com a natureza distribuída e multi-consumidores de SOA.
	Refatorações que não modificam interface ou comportamento agregam valor ao negócio e aumentam a qualidade do código sem gerar impacto a consumidores.
	No entanto, haverá casos onde mudanças no negócio ou em requisitos não-funcionais requeiram modificações de comportamento e/ou de contrato. Por exemplo, um requisito legal pode exigir mudanças imediatas na forma de autenticação de um serviço, onde um dado *token* deve passar a ser informado a cada requisição. Neste caso, será inevitável

	modificar o contrato e impactar consumidores. Haverá, no entanto, casos onde uma mudança no contrato pode ser tratada de forma mais gradual. Nestes casos, será possível lançar mão de estratégias de *design* onde um estágio intermediário e temporário seja usado para garantir a compatibilidade retroativa. Isso permitirá um período de transição entre o novo comportamento e o descontinuado até que todos os consumidores sejam migrados, sem impacto imediato aos mesmos.
	Não há referências, em nosso estudo de mapeamento sistemático, indicando que o uso de XP leve a uma maior incidência de refatoração de contratos. Conforme CALLAHAN [**?**], cada serviço é visto como uma funcionalidade, de forma análoga ao desenvolvimento de sistemas "tradicionais". Uma vez identificada a funcionalidade, esta deriva uma *User Story*, e respectivos critérios de

aceitação. Tais critérios guiarão o *design* e implementação do serviço, conforme discutido acima. Uma modificação radical do contrato só será necessária se houver uma mudança radical no negócio, que impacte a definição original da *User Story* ou de seus critérios de aceitação. Neste sentido, do ponto de vista de gestão e refatoração de contratos, não há diferença,entre uma abordagem “tradicional” e uma abordagem XP, isto é, o contrato só é modificado se houver mudança nas necessidades da organização ou consumidores do serviço.
Além disso, em um cenário onde times distribuídos trabalham paralelamente em funcionalidades com pontos de interseção, a definição antecipada do contrato, ainda que em uma versão mínima e posteriormente refatorada, ganha ainda mais importância. Ela possibilita, por exemplo, que os provedores e

	consumidores do contrato trabalhem sem dependências funcionais entre si, usando ferramentas de *mock* para simular o comportamento esperado do serviço que expõe o contrato. Por exemplo, num cenário onde o time P trabalhará no desenvolvimento da parte "provedora" de um serviço de forma paralela ao time C, responsável pelo consumo do serviço, uma vez que o contrato esteja definido, o time C pode usar ferramentas como o soapUI [3] para simular o comportamento do serviço sendo desenvolvido pelo time P. Considere que o serviço disponibiliza a operação: *+ recuperarNomePorCpf (String cpf): String* Por meio de ferramentas de simulação, o time C pode criar um *mock* do serviço, programando e simulando o comportamento do serviço em resposta a diferentes entradas. Por

[3] http://www.soapui.org/

exemplo, o time C pode programar o *mock* para retornar uma mensagem de “CPF inválido” para a entrada “000.000.000-00”; ou retornar o nome “João Alberto da Silva” para o CPF “012.345.678-90”. O time C tem controle total sobre a massa de dados necessária a seus testes, e o time P não precisa se preocupar com o impacto que seria causado ao time C se o serviço fosse tirado do ar com frequência, nem em fornecer respostas adequadas aos testes do time C.
Se o contrato for modificado, basta que os times sejam comunicados da necessidade de mudança para encaixar a modificação da melhor forma em seu trabalho. O time C pode esperar um momento mais oportuno para encaixar a mudança em seu planejamento, enquanto o time P não precisa se preocupar em disponibilizar o serviço atualizado o mais rápido possível de forma a minimizar o impacto sobre o time C. Ambos os times podem

	trabalhar em paralelo sem outras dependências que não a definição do contrato.
Componente / atividade a que se refere:	Projetar nível de acoplamento
Considerações:	Sobre os tipos de acoplamento propostos por ERL [4], considere os exemplos a seguir **(a)**. Ao realizar o design de um serviço, os membros da equipe responsáveis por esta tarefa devem procurar maximizar o acoplamento da **lógica ao contrato**, de forma que a lógica seja a resposta mais simples ao contrato estabelecido. Da mesma forma, os membros devem visar o mínimo acoplamento do **contrato a lógica**, de forma a garantir que, caso a lógica do serviço se modifique, o contrato se mantenha estável e os consumidores tenham impacto mínimo. Isso pode ser alcançado, por exemplo, garantindo que a lógica implementada seja a resposta mais

	simples à necessidade de negócio, evitando, no entanto, referências a nomes de tabelas ou componentes internos. O baixo acoplamento do **contrato à tecnologia** pode ser usado omitindo no contrato referências a componentes ou protocolos de terceiros. Da mesma forma, o baixo acoplamento do **contrato à implementação** pode ser alcançado garantindo que esta seja a resposta mais simples à necessidade do negócio, sem referências a caminhos de rede, versões de API's ou qualquer outra informação inerente ao ambiente de execução do serviço. Por fim, o baixo acoplamento do **contrato a funcionalidades** externas pode ser alcançado projetando-se o serviço de forma a ser a resposta mais simples à necessidade de negócio de forma agnóstica ao contexto, sem explicitar no contrato, por exemplo, referências a uma funcionalidade externa reutilizadas pelo

	serviço no cumprimento de sua função. Por sua vez, o tratamento dos tipos de acoplamento definidos por KRAFZIG [29] se refere a fases e papéis diversos, ao longo de um ciclo completo de desenvolvimento de *software*. Nem todos serão tratados por este trabalho, uma vez que este destina a tratar especificamente a fase de construção. De forma geral, os pontos passíveis de tratamento, no que se refere aos tipos de acoplamento definidos por KRAFZIG [29] se relacionam a critérios de aceitação ou políticas de governança: **Conexões físicas**: a escolha da forma de conexão física pode não ser uma decisão da equipe. Por exemplo, se a organização onde o serviço será implantado não dispuser de mecanismos de mediação (por exemplo, barramento de serviços), só será possível fazer uso de conexões ponto a ponto, ainda que sob pena de

	aumentar o acoplamento da solução. Por outro lado, se há um mecanismo de mediação estabelecido pela organização, e uma política de utilização do mesmo, então esta informação deveria constar no critério de aceitação em forma de requisito não-funcional, de forma que o produto da iteração siga a definição do PO.
	Estilo de comunicação: O estilo de comunicação entre serviços e os consumidores dos serviços pode não ser uma decisão da equipe, pois permeia questões de desempenho, experiência de usuário e governança. Por exemplo, se o desempenho do serviço se deteriora de acordo com os parâmetros da requisição, mas essa deterioração produz respostas em um tempo aceitável para o PO sem comprometer a disponibilidade do serviço, então o PO pode fazer constar nos requisitos funcionais que a comunicação deve ser síncrona. Analogamente, se houver estabelecida uma diretriz onde os serviços devem

preferencialmente ser assíncronos, esta deve constar dos critérios de aceitação. O serviço produzido será sempre a forma mais simples de atender aos critérios.

Modelo de dados: conforme BECK [16], o produto da construção será a implementação mais simples que atende aos critérios de aceitação. Desta forma, ao utilizar-se práticas XP, a tendência é atender naturalmente a prática de "simple common types only", se esta for a forma mais simples de entregar o produto funcionando.

Quando se fala de tipos de dados, segundo JOSUTTIS [3], intrinsecamente está se falando de harmonização de tipos em uma corporação. É comum uma organização ter dificuldades em garantir que todos os seus sistemas usem o mesmo modelo canônico de objetos de negócio, pois: (i) os seus sistemas evoluem em tempos diferentes; e (ii) os responsáveis por cada sistema têm uma visão diferente sobre um conceito do negócio. Estes fatores levam a sistemas

que se comunicam por meio de
representações diferentes do mesmo
objeto de negócio.
Por exemplo, a área de marketing de uma
corporação pode ter uma visão de
que um cliente possui os atributos
nome, endereço eletrônico e
telefone, os quais são utilizados nos
contratos dos serviços de vendas. A área
de cobrança, por outro lado, pode se
preocupar mais com informações como CNPJ
e endereço comercial de um cliente.
Unificar essas visões em uma única
representação é tarefa das mais árduas e
que tende a não se concretizar
[3].
Além disso, o estabelecimento de um
modelo canônico de objetos de
negócio gera acoplamento dos sistemas
a uma representação do modelo
de negócio, o que força a atualização
constante de todos os sistemas, toda vez
que uma modificação neste modelo
canônico é publicada. Desta forma, a
recomendação de JOSUTTIS [3] apud

	KRAFZIG [29] é que as estruturas de dados sejam as mais simples possíveis, permitindo menor acoplamento entre os sistemas de uma corporação. Note-se que isso não significa dar preferencia a tipos simples (por exemplo, *String*, *Integer*, *Boolean*) como parâmetros de serviços. Ao contrário, esta recomendação diz respeito a não dedicar muito esforço na tentativa de se definir um modelo de comunicação complexo, tentando prever e contemplar todas as visões possíveis sobre um objeto de negócio.
	Tipagem: este ponto está bastante relacionado às práticas de *Simple Design* e *Refactoring*, além de critérios de aceitação. A equipe terá como objetivo entregar o produto mais simples que atende aos critérios de aceitação. Logo, se não houver no *Backlog* uma definição razoavelmente imutável de outras funcionalidades que requeiram uma flexibilização dos tipos de dados, e nem houver critérios de

aceitação indicando o uso de tipos fracos, é provável que a equipe, via de regra, use tipos fortes, por serem de mais fácil leitura e interação. Conforme a iteração evolui, a equipe aprende mais sobre o escopo e o uso de tipos de dados fracos pode se tornar necessário, de modo a facilitar o reuso. Neste caso, *Refactoring* é usado para flexibilizar os tipos de dados.

Tipos fortes são aqueles que requerem validação de sua integridade. Por exemplo, se uma dada informação é definida como numérica, não deve ser possível atribuir a ela o valor "essa é uma string". Analogamente, uma informação fracamente tipada é mais genérica e dispensa validação de seu conteúdo. Quando se fala de tipos fortes ou fracos, estão envolvidos fatores como tempo de processamento, acoplamento, e evolução.

Conforme JOSUTTIS [3], tipos fortes são de mais fácil processamento e mais fácil de se trabalhar pelo

	programador. Por outro lado, uma informação fortemente tipada precisa ser validada (no caso de XML, a validação é feita via *schemas* XSD), o que demanda tempo de processamento. Além disso, considere o caso onde a representação de um parâmetro como CPF, onde um sistema pode retornar como um número e outro como string. Nesse caso, ao invés de optar pelo uso de tipagem forte de um parâmetro como CPF, a qual pode falhar com frequência, o uso de tipagem fraca pode ser mais aconselhável, dispensando processamento na validação e permitindo a comunicação pela forma considerada mais adequada pelas partes. Além disso, se uma informação sujeita a mudanças de tipo como CPF é definida fortemente tipada como valor numérico, por exemplo, e um serviço precisar ser evoluído e o tipo do CPF alterado para string, o acoplamento entre os consumidores e o serviço é tão alto que o custo da mudança pode inviabilizá-la.

	Padrão de interação: de forma análoga a *Data Model*, se não houver definições contrárias, a equipe procurará entregar o que for mais simples e mais legível. Considerando que *self-contained messages* são menos complexas e mais inteligíveis, essa será a tendência. O conceito de *self-contained messages* está relacionado a complexidade de construtos. Segundo JOSUTTIS [3], quanto mais complexa a definição de um objeto de negócio e quanto menos tipos de dados fundamentais (*String*, *Integer*, *Boolean*, *etc.*) são usados em sua definição, maior o risco de se deparar com problemas de portabilidade. Por exemplo, o uso do tipo *List* em detrimento de arrays em serviços construídos em Java tipicamente representa um risco de portabilidade. Linguagens como, por exemplo, C#, tem dificuldades de interagir com coleções que representam *List*, ao passo que o uso de *Array* para coleções é aceito nativamente.

	JOSUTTIS [3] recomenda que, de forma geral, construtos customizados sejam endereçados de forma conservadora, preferindo objetos menos complexos e que façam uso extensivo de tipos de dados fundamentais.
Componente / atividade a que se refere:	Projetar abstração
Considerações:	Um exemplo de exposição de detalhes não-funcionais **(a)** seria o caso onde, em não havendo nenhum requisito funcional ou não-funcional regulando o aspecto de acoplamento por conexões físicas, a equipe poderia optar por acordar com o PO uma regra onde, em horários de pico, para requisições superiores a um dado *threshold*, a resposta será enviada assincronamente por e-mail ao usuário que invocou a funcionalidade. No que se refere às quatro dimensões de abstração propostas por ERL [4], pode se lançar mão, por exemplo, da revisão em pares com o propósito de identificar uma violação de **abstração tecnológica** pela

	exposição no contrato do detalhe tecnológico de que o serviço foi implementado na plataforma JEE, ou uma violação de **abstração lógica**, dada pela informação de que uma funcionalidade ordena seus resultados usando o algoritmo *Bubble Sort*. Por outro lado, informações de potencial de reuso como as utilizadas no componente APRQ.ABST.0004 podem ser úteis na avaliação da **abstração funcional**. Por exemplo, se a funcionalidade sendo trabalhada na iteração 1 vai executar uma lógica que também será necessária a outra funcionalidade prevista mais a frente no *Backlog*, pode ser interessante expor esta funcionalidade no contrato imediatamente, mesmo que ainda não haja consumidores naquele momento.
Componente / atividade a que se refere:	Projetar reuso
Considerações:	Analogamente ao conceito **(a)**, de acordo com KROGDAHL, LUEF, & STEINDL [10], todo serviço deve ser considerado

	potencialmente reutilizável por outras aplicações. No entanto, a interface de um serviço evolui com o tempo, conforme a organização melhora seu entendimento sobre as responsabilidades dos serviços. O uso de *Refactoring* constante para atender a esta evolução não pode ser ignorada ou evitada, sob pena de gerar um modelo de serviços inflexível a longo prazo. Este modelo pode não suportar um ambiente de negócios sob constantes mudanças. Além disso, segundo NARAYANAN [70], o uso de *Refactoring* permite eliminar duplicidades entre serviços, seja por refatorações internas a um serviço, ou pelo reuso de um serviço pré-existente. Assim, os critérios de aceitação são uma ferramenta importante em conjunto com *Refactoring*, uma vez que este conjunto garante que o comportamento do serviço modificado continua aderente às necessidades da organização.
Componente /	Projetar conectores / adaptadores

atividade a que se refere:	
Considerações:	Por exemplo, se o ativo legado **(a)** espera uma coleção de entradas, mas o novo serviço só trabalha com uma entrada, será necessário preparar uma lógica que envie ao ativo legado uma coleção contendo uma única entrada.
Componente / atividade a que se refere:	Projetar autonomia
Considerações:	No que se refere a preocupações com ambiente **(a)**, de forma complementar, as práticas de *Testing* e *Continuous Integration* reforçam a eliminação de dependências externas por meio de testes unitários executando em um ambiente neutro. Como o código tem de ser projetado para passar nos testes, e os testes rodam em ambiente neutro, os testes falharão se o código tiver alto acoplamento com dependências externas. Em conjunto, *Refactoring* é usado ao longo do tempo para diminuir o acoplamento entre o serviço e suas

	dependências.
Componente / atividade a que se refere:	Projetar ausência de estado
Considerações:	O projeto e implementação deste princípio são facilitados por capacidades das plataformas onde executam. A plataforma JEE, por exemplo, provê mecanismos de armazenamento de estado a serviços desenvolvidos em Java, ativados por meio de *Annotations*. Por exemplo, a annotation *@Stateful* ativa a oferta do mecanismo de gerenciamento de estado para *Web Services* construídos em Java. Apesar de ERL [4] recomendar que serviços sejam *Stateless* sempre que possível, de modo a maximizar sua escalabilidade, as necessidades funcionais de um dado serviço podem tornar mandatório lançar mão de algum mecanismo adicional de gerenciamento de estado. A suíte de testes serve de ferramenta para indicar se há retenção de estado no serviço, uma vez que a dependência entre os testes pode ser um

	indicativo de retenção indevida de estado. Além disso, com os testes executando a cada integração, garante-se que o comportamento esperado permanece inalterado a cada modificação do código, conforme recomendado por BECK [16]. Neste caso, o comportamento que espera-se ser mantido é a retenção mínima de estado.
Componente / atividade a que se refere:	Projetar descoberta de serviço
Considerações:	O uso de *Coding Standards* para a exposição das capacidades **(a)** no contrato favorece contratos de melhor descoberta e interpretação [4]. Além disso, o uso de uma Metáfora que seja de entendimento comum a PO e equipe de desenvolvimento e que represente adequadamente um ou mais conceitos de negócio pode favorecer o entendimento do propósito do serviço.
Componente / atividade a que se refere:	Planejar tarefas
Considerações:	O delineamento coletivo do trabalho a

	ser feito **(a)** é uma ferramenta de comunicação, que visa equalizar entre todos os membros o entendimento do que precisa ser feito para atingir os critérios de aceitação definidos para a funcionalidade. A designação de responsáveis pelas tarefas é feita de forma voluntária, ou seja, não há uma figura central responsável por distribuir tarefas, cada par de membros da equipe se candidata a resolver uma tarefa por vez. Exemplos de tarefas a serem desenvolvidas pela equipe podem ser: "definir contrato da funcionalidade", "escrever testes de aceitação não-funcionais", "executar testes de sistema", "disponibilizar funcionalidade para PO", entre outros.
Componente / atividade a que se refere:	Atribuir responsabilidades
Considerações:	Quando o trabalho se inicia **(a)**, haverá tarefas sem responsáveis assinalados, uma vez que pode haver tarefas encadeadas ou um número insuficiente de

membros na equipe para trabalhar em
todas as tarefas em paralelo.
Conforme o trabalho progride e novas
tarefas são assinaladas **(b)**, o
progresso do desenvolvimento é medido
visualmente, pela proporção de tarefas
resolvidas em relação ao número de
tarefas planejadas.
Eventualmente, a alocação de tarefas
pode seguir algum critério estratégico.
Por exemplo, de forma a obter maior
nivelamento de conhecimento e
habilidades entre os membros da
equipe, pode ser interessante que os
membros mais experientes trabalhem em
par mais frequentemente com os membros
menos experientes. KARSTEN & CANNIZZO [15]
citam a rotatividade de responsabilidades
como fator de motivação para equipes.
Em situações críticas, pode ser
interessante usar pares de pessoas
mais experientes para resolver
problemas urgentes, enquanto os
demais trabalham em itens menos
prioritários. A distribuição dos pares é

	decidida pela equipe em comum acordo entre todos. O planejamento e evolução das tarefas podem ser registrados em qualquer meio. Na literatura há registros de equipes usando tanto quadros com *post-it's* quanto ferramentas online de gerenciamento de projetos ágeis. Restrições de mídia para esse propósito devem ser acordadas entre os envolvidos no projeto.
Componente / atividade a que se refere:	Definir formato dos testes de integração
Considerações:	Quanto às diferenças, do ponto de vista de teste, entre aplicações ditas "tradicionais" e aplicações orientadas a serviço, pode-se citar como exemplo o fato de as ligações dinâmicas entre serviços SOA poderem tornar impossível prever o resultado exato retornado por uma composição de serviços. Há diversos exemplos de fatores que podem influenciar na decisão sobre a forma dos testes de integração. Considere o caso onde há a necessidade de reutilizar um

	serviço provido por um fornecedor externo, e cada invocação do serviço é cobrada ou há um limite mensal de invocações por cliente. Nesse caso, se for criado um teste de integração que invoca o serviço do fornecedor diretamente, e este teste for executado em toda construção disparada no ambiente de integração contínua, com o tempo este teste pode se revelar financeiramente custoso, ou consumir rapidamente o limite de invocações franqueado à organização. Outro exemplo é o de uma organização com orçamento restrito, onde não há orçamento para a implantação de um ESB, e que trabalha com ligações (*binding*) ponto-a-ponto entre serviços. Nesse caso, se o serviço reutilizado sair do ar, não haverá outro serviço disponível que possa dinamicamente substituí-lo, fazendo com que a suíte de testes falhe sem a possibilidade de intervenção da equipe que desenvolve o serviço "principal". O termo API **(a)** se refere a uma biblioteca que provê o

	isolamento entre consumidor e provedor. Essa pode ser uma alternativa escolhida pela organização para incentivar o reuso de um serviço. Por exemplo, considere um serviço de alto índice de reuso em uma organização e que tem requisitos específicos de segurança que demandam alguma complexidade do lado do consumidor para estabelecer uma conexão segura. Ao invés de toda equipe consumidora do serviço implementar a mesma lógica complexa de estabelecimento de conexão segura, a parte provedora poderia fornecer uma API que encapsulasse essa lógica complexa, de forma que os consumidores só se preocupassem com o uso da lógica de negócio do serviço, deixando a complexidade de conexão a cargo desta API. No caso de Java, esta API poderia ser fornecida como um arquivo .JAR, incluído como dependência de todo consumidor. Desta forma, se houver mudança na forma de conexão com este serviço, a organização não precisa se

	preocupar com impacto direto a todos os consumidores. Basta atualizar esta API e disponibilizar a atualização da mesma, restando um impacto razoavelmente menor a todos os consumidores, uma vez que estes consumidores não se comunicam diretamente com o serviço em si, mas sim com a API que estabelece a conexão. A técnica de simulação de funcionamento via *mock* **(b)** tem o benefício de não requerer ferramental externo, como, por exemplo, um ESB. A comunicação entre provedor e consumidor fica sob controle do mesmo time, sem a necessidade de preparação de massas de dados, ou acordos com outras equipes ou fornecedores. Além disso, no caso de cobrança pela invocação de serviços externos, essa alternativa acarreta cobrança zero, uma vez que a invocação do serviço externo não se completa. Por outro lado, a simulação de um serviço acarreta menor confiabilidade sobre o funcionamento integrado da solução como um todo. Por exemplo, se um serviço

	reutilizado tem seu contrato ou lógica modificados, ou simplesmente saiu do ar, um teste de integração por simulação não será capaz de avisar aos envolvidos no projeto de que houve quebra de compatibilidade ou indisponibilidade. Para casos onde é importante averiguar o funcionamento real da integração entre os serviços, propomos duas alternativas, de acordo com a forma como a ligação entre os serviços é estabelecida. Em ambos as alternativas, nossa proposta é o uso de serviços simulados, ao invés de simulações em nível de API. Neste caso, garante-se o funcionamento do canal de comunicação e a disponibilidade do serviço reutilizado (representado de forma simulada), mas há ainda pendentes as questões de mudança de contrato ou lógica. A diferença entre as alternativas se refere a operacionalização das mesmas. Quando as ligações entre serviços são estabelecidas ponto-a-ponto, será necessário disponibilizar um serviço

mock, acessível durante a execução dos testes e com uma massa de dados estável e representativa, com informações que atendam aos cenários cobertos pelos critérios de aceitação. Além disso, durante a execução dos testes, o endereço do endpoint deve ser dinamicamente modificado, para apontar para o serviço simulado. Quanto a acordos com provedores de serviço **(c)**, alguns exemplos de pontos passíveis de acordo são: um ambiente segregado para execução de testes integrados, uma massa de dados estável, uma faixa de dados imutável que atende aos critérios de aceitação. Num caso onde o serviço sendo desenvolvido depender de uma composição de serviços, se não houver necessidade de precisão na verificação do retorno do serviço, a equipe deverá traçar uma estratégia de validação do resultado final retornado por esta composição. Por exemplo, no caso da companhia aérea fictícia, com um serviço de busca de voos disponíveis dentro de

	uma faixa de datas, uma possível estratégia é verificar se todos os voos decolam dentro da faixa de datas escolhidas, ao invés de verificar se os números dos voos são os esperados.
Componente / atividade a que se refere:	Revisar código
Considerações:	Dependendo da quantidade e complexidade das violações encontradas **(a)**, a equipe, em conjunto com o PO, pode definir a melhor estratégia de correção das violações. Por exemplo, num caso onde seja inviável corrigir todas as violações sem comprometer o andamento da iteração, a equipe de desenvolvimento e PO podem acordar pelo registro das violações pendentes em ferramenta acordada, sob a denominação "débito técnico" [4]. Estas violações podem, então, ser priorizadas e corrigidas no momento adequado. Esta priorização do débito técnico pode obedecer a critérios variados. Uma possível estratégia é

[4] http://martinfowler.com/bliki/TechnicalDebt.html/

	encaixar a correção de violações nas atividades da *Sprint* sempre que o seu escopo for completado com antecedência de alguns dias em relação a entrega da iteração. Outra estratégia é, caso o andamento da iteração esteja dentro do planejado ou adiantado, definir diariamente uma pessoa para ficar concentrada em resolver violações, diminuindo assim, gradualmente, o tamanho do débito técnico da equipe. Caso haja número excessivo de violações, é importante que a equipe se reúna para discutir e tentar chegar a causa raiz deste problema, sob pena de voltar a acontecer e seu débito técnico crescer exponencialmente.
Componente / atividade a que se refere:	Disponibilizar serviço
Considerações:	Segundo ERL [4], um serviço deve, por princípio, expressar por meio de um contrato **(a)** seu propósito, capacidades, restrições e informações semânticas que o responsável pelo serviço deseje tornar

públicas. Estas informações são descritas seguindo normas de *design* compartilhadas por todos os serviços contidos em um dado repositório. Este contrato pode ser expressado por meio de um conjunto de documentos de descrição do serviço, ficando cada um responsável por descrever uma parte do serviço. Este conjunto é composto por documentos a serem consumidos em tempo de execução e/ou documentos textuais, "não-técnicos", que complementam seus detalhes técnicos. Há diversas formas de se expressar um documento técnico de descrição de serviço. Independente do formato, segundo ERL [4], os objetivos do contrato são:

- aumentar o nível de interoperabilidade entre serviços de um mesmo repositório, por meio de modelos de dados consistentes, o que reduz a necessidade de transformações de dados;
- permitir que o propósito e as capacidades do serviço sejam facilmente inteligíveis, favorecendo sua

	interpretação e previsibilidade; De forma geral, os objetivos acima reforçam que um contrato se destina a leitura e interpretação tanto de máquinas quanto de seres humanos. Mais do que simplesmente favorecer o entendimento humano do detalhamento textual de suas capacidades e restrições, estes objetivos visam facilitar a automação da descoberta e uso de um dado serviço, perseguindo os objetivos finais de maximização do reuso e minimização de desenvolvimento customizado para atender a um objetivo de negócio.
Componente / atividade a que se refere:	Adequar princípios de SOA
Considerações:	De forma análoga a revisão de código, a quantidade e complexidade das violações encontradas definirão o melhor momento para se executar suas correções **(a)**. Caso não seja viável tratar estas violações imediatamente, estas devem ser tratadas como débito técnico. Ver as considerações sobre o componente *Revisar*

	código para maiores informações sobre estratégias e melhores práticas para tratamento do débito técnico de uma equipe.
Componente / atividade a que se refere:	Acompanhar andamento da iteração
Considerações:	O Mega Framework [20] **(a)** foi criado a partir da experiência dos autores com a implantação do Scrum em sua organização. Esta adoção se iniciou com três projetos pilotos, todos contidos em um mesmo produto: dois bem sucedidos, de menor escopo, e um terceiro, composto de um time de 15 pessoas, onde houve problemas de prazo na entrega e foi percebido gasto excessivo de tempo em discussões e análises, além de problemas de conflitos pessoais entre os envolvidos no projeto. Observou-se que um time com tantas pessoas, o que é contrário às recomendações do Scrum [57] poderia ser uma das causas do problema. A partir daí, essas pessoas foram divididas em times menores, cada qual

	responsável por um conjunto de funcionalidades relacionadas, e o Mega Framework foi sendo criado, de forma a endereçar os problemas de comunicação inerentes a times trabalhando em paralelo no mesmo produto. O *framework* propõe estratégias e reuniões periódicas, as quais visam, em última instância, manter a comunicação fluindo em diversos níveis: interna a uma equipe, entre equipes, entre PO's, *etc*. No que se refere a estratégias, foi definido, por exemplo, que novos times não seriam criados utilizando somente pessoas recém-contratadas: os novos times seriam formados por uma combinação de novos contratados e pessoas com experiência no produto, de forma a facilitar a adaptação dos novos e facilitar a comunicação entre os diferentes times. Foi também definido que os times seriam divididos por funcionalidade, ou seja, cada time seria responsável por um conjunto de

	funcionalidades afins, de forma que a quantidade de informação sob responsabilidade de cada time fosse mais administrável. Foi também definido que todos os times utilizariam iterações com um mês de duração, onde toda semana haveria entregas de times diferentes, facilitando a organização das agendas dos interessados, entre eles, os diversos PO's e clientes. No que se refere a reuniões, o Mega Framework propõe 8 tipos, relacionadas a diferentes fases do ciclo de vida de desenvolvimento, conforme sumarizado na Tabela 8.2. A LPS proposta só utiliza duas das práticas preconizadas pelo Mega Framework. As demais práticas, ainda que válidas, não se direcionam ao acompanhamento do andamento da iteração durante a fase de construção. Por exemplo, a *Weekly Pre-Planning* é conduzida pelos PO's, com o objetivo de antecipar o detalhamento dos itens candidatos a

	entrarem no *Backlog* da próxima iteração. Uma vez que esta prática é de responsabilidade única dos PO's e diz respeito a definições de negócio, escapa às as responsabilidades da equipe durante a fase de construção. Analogamente, a prática *Regular Mega Meetings with Business Area* é conduzida entre PO's e gerentes de áreas operacionais, com o objetivo de identificar novas oportunidades para o produto, fugindo ao escopo da fase de construção.

Tabela 8.2: Reuniões previstas no Mega Framework [20]

Reunião	**Propósito**	**Fase relacionada do ciclo de vida**	**Participantes**
Mega Planning	Ocorre após os planejamentos de iteração de cada time, e serve para disseminar o que foi planejado para cada iteração de cada time,	Construção	Representantes de cada time

	identificando potenciais conflitos ou relacionamentos entre itens de *Backlog*.		
Mega Stand-Up	Ocorre após decorrida metade da iteração, tem por objetivo verificar se todos os times estão com andamento dentro do planejado e/ou discutir os impedimentos que atrapalham o andamento de cada equipe.	Construção	Representantes de cada time
Mega Retrospective	Ocorre a cada seis meses com o objetivo de identificar impedimentos que prejudicam o andamento de mais de um time.	Pós-Construção	Representantes de cada time
Sprint Review	Ao final de cada Sprint, o time apresenta a diversos interessados os resultados alcançados e as funcionalidades entregues	Pós-Construção	PO, clientes, interessados e time

	pelo time.		
Weekly Pre-Planning	Tem por finalidade detalhar os itens do backlog priorizados para a próxima iteração e identificar que itens devem ser postergados até que tenham maior riqueza de informações para detalhamento	Planejamento	PO's e interessados
Weekly Product Owner & Scrum Master Meeting	Visa discutir impedimentos, funcionalidades importantes planejadas e que afetam múltiplos times, além de tomadas de decisão. tenham maior riqueza de	Planejamento	PO's, Scrum Masters
Regular Mega Meetings with Business Areas	Têm por objetivo discutir mensalmente pendências, oportunidades de negócio e problemas encontrados pelos times durante o andamento das iterações.	Planejamento	PO's, gerentes de áreas operacionais
Knowledge Sharing	Visa disseminar boas práticas e troca de	Externo ao ciclo de	PO's, gerentes de

	conhecimento.	desenvolvimento	áreas operacionais

9. Apêndice D - Mapeamento de práticas ágeis

Este apêndice apresenta o mapeamento de práticas compartilhadas entre métodos ágeis, conforme publicado em CARVALHO & AZEVEDO [17].

Apesar de originalmente estes resultados haverem sido apresentados em uma só tabela, por questões de limitação de espaço, os resultados foram divididos nas duas tabelas a seguir. Os percentuais na segunda linha de cada tabela representam quantas práticas são utilizadas por aquele método, dentro do universo de práticas ágeis catalogadas.

	DSDM	**Crystal**	**RUP**	**XP**	**ASD**	**Scrum**
Specific / shared properties	**11.5%**	**13.5%**	**7.7%**	**44.2%**	**5.8%**	**15.4%**
Continuous delivery	x	x		x		
Fitness for business purpose is the essential criterion for acceptance of deliverables	x		x	x		
All changes during development are reversible	x					
Requirements are baselined at a high level	x					
Testing is integrated throughout the lifecycle	x			x		
MoSCoW Rules	x					
Automated acceptance testing		x		x		

Two user viewings per release		x				
Managed requirements			x	x		x
Component-based architecture			x		x	
Visual modeling			x			
Metaphor				x		
Simple design				x		
Refactoring				x		
Pair programming				x		
Collective ownership				x		
Continuous integration				x		
Sustainable pace				x		
Coding standards				x		
Mission oriented					x	
Risk-driven					x	
Product backlog						x
Effort estimation				x		x
Sprint planning		x		x		x
Sprint backlog		x				x
Dailiy Scrum meeting						x
Sprint review		x				x

Take responsibility for what you do				x		
Don’t put up with bad design or code				x		
Take an active role in introducing change where you feel it’s necessary				x		
Constantly broad your knowledge						
Improve your communication skills						
Develop in parallel						
Depend on tools						
Establish a stable architecture						
Ignore maintenance						
Tailor the methodology daily						
Accept multiple valid approaches						
Domain object modeling				x		
Developing by feature						
Individual class ownership						
Feature teams						
Inspection				x		
Configuration management						
Progress reporting		x				

Built by volunteers						
Work is not assigned				x		x
There is no explicit system level design						
There is no project plan, schedule or list of deliverables						
Eliminate waste						
Decide as late as possible				x		
See the whole				x		

	PP	**ISD**	**AM**	**FDD**	**OSSD**	**Lean**
Specific / shared properties	**11.5%**	**15.4%**	**3.8%**	**17.3%**	**13.5%**	**13.5%**
Continuous delivery		x				x
Fitness for business purpose is the essential criterion for acceptance of deliverables			x	x	x	x
All changes during development are reversible						
Requirements are baselined at a high level						
Testing is integrated throughout the lifecycle					x	
MoSCoW Rules						
Automated acceptance testing					x	

Two user viewings per release						
Managed requirements	x					
Component-based architecture		x				
Visual modeling						
Metaphor						
Simple design			x			
Refactoring						
Pair programming						
Collective ownership						
Continuous integration				x		
Sustainable pace						
Coding standards						
Mission oriented						
Risk-driven						
Product backlog						
Effort estimation						
Sprint planning						
Sprint backlog						
Dailiy Scrum meeting						
Sprint review						

Take responsibility for what you do	x					
Don’t put up with bad design or code	x					
Take an active role in introducing change where you feel it’s necessary	x					
Constantly broad your knowledge	x					x
Improve your communication skills	x					
Develop in parallel		x				
Depend on tools		x				
Establish a stable architecture		x				
Ignore maintenance		x				
Tailor the methodology daily		x				x
Accept multiple valid approaches		x				
Domain object modeling				x		
Developing by feature				x		
Individual class ownership				x		
Feature teams				x		
Inspection				x		
Configuration management				x		
Progress reporting				x		

Built by volunteers					x	
Work is not assigned					x	
There is no explicit system level design					x	
There is no project plan, schedule or list of deliverables					x	
Eliminate waste						x
Decide as late as possible						x
See the whole						x

10. Apêndice E - Exemplo de uso da LPS

Este apêndice apresenta um exemplo de uso da LPS proposta, a partir de uma instanciação da mesma para atendimento a em um cenário fictício. Este cenário foi elaborado com o propósito específico de ilustrar os conceitos apresentados nesta dissertação.

10.1Cenário

A empresa SmartBrick desenvolveu uma metodologia que suporta o fornecimento de indicadores de produtividade em obras de engenharia civil.

Esta metodologia consiste simplesmente da aferição das atividades desempenhadas pelos trabalhadores de uma obra em um dado momento.

Por exemplo, considere uma obra que está construindo um prédio residencial, e a obra está numa fase intermediária. Num determinado intervalo de uma hora de duração, um fiscal de campo passa pela obra e verifica que há 2 pessoas levantando uma parede A, 3 pessoas pintando uma parede B, 1 pessoa limpando a sala X, e 2 pessoas esperando que mais cimento seja disponibilizado para sua tarefa. Nesse caso, o relatório desse fiscal indicará que, naquele horário, havia 5 pessoas efetuando atividades diretamente atreladas a entrega da obra (2 levantando uma parede e 3 pintando uma parede) e 3 pessoas ociosas ou executando tarefas secundárias, que não contribuem com a entrega da obra. Ou seja, o indicador daquele horário apontará que, das 8 homens-hora aferidas, 37.5% foram desperdiçadas. Com uma verificação sendo desempenhada continuamente ao longo de um dia de trabalho, chega-se a um indicador de percentual de horas desperdiçadas ao longo do dia e/ou do mês de trabalho.

O mercado de construtoras, ao conhecer tal metodologia, demonstrou grande interesse em seu uso, visando aumento de sua margem de lucro e melhor aproveitamento de sua força de trabalho. Com isso, a SmartBrick decidiu contratar uma consultoria para trans-

formar essa metodologia em um produto.

A consultoria XSOA foi contratada, e as empresas acordaram pela construção do produto utilizando a LPS proposta nesta dissertação. A SmartBrick, interessada num produto com a maior robustez e qualidade possível, se dispôs a fornecer todo o suporte a instanciação da LPS, oferecendo uma pessoa com grande conhecimento da metodologia, articulada e agregadora, com a responsabilidade de desempenhar o papel de PO. Esta pessoa estaria disponível sempre que necessário, fornecendo as informações necessárias do ponto de vista do negócio, e servindo de agregador das demandas de negócio, bem como esclarecimento de dúvidas junto a outros funcionários da SmartBrick. A XSOA também solicitou que essa pessoa estivesse fisicamente disponível no escritório da XSOA ao longo do desenvolvimento, mesmo que em tempo parcial, para esclarecer dúvidas da equipe da XSOA. Foi acordado que essa presença seria maior nas primeiras iterações, sendo diminuída progressivamente conforme o progresso do projeto, uma vez que a expectativa de ambos as empresas era que a equipe fosse absorvendo os conceitos com o tempo e demandando menos suporte contínuo.

10.2Produto

Durante o entendimento do projeto e levantamento de requisitos, chegou-se a conclusão que o produto a suportar a metodologia deveria ser composto de dois componentes principais: um dispositivo portátil onde o fiscal de campo pudesse registrar seus apontamentos; e, uma aplicação web, que disponibilizaria relatórios com os indicadores de produtividade aos diversos clientes da SmartBrick. A comunicação entre o dispositivo portátil e a aplicação web seria feita através de um serviço. Uma vez que era possível não haver cobertura de rede em algumas obras, a SmartBrick também definiu que o aplicativo móvel deveria poder ser usado de forma off-line, isto é, um fiscal de campo deveria poder registrar as informações ao longo de todo o dia de trabalho, ficando responsável por executar a sincronização dos dados ao fim do dia.

Dadas as características do produto e questões de alocação de equipe, a XSOA decidiu alocar três equipes: a equipe **W**, localizada na mesma cidade da SmartBrick ficou responsável pelo desenvolvimento da aplicação web; a equipe **S**, na mesma cidade, mas em escritório diferente da equipe W, ficou com a responsabilidade de desenvolver os serviços; a equipe **D**, localizada em outra cidade, ficou responsável pelo desenvolvimento do aplicativo para o dispositivo portátil.

10.3Requisitos

De acordo com o PO, os requisitos básicos da solução como um todo eram razoavelmente simples:

R1 – Deve ser possível configurar as atividades que demandam aferição em uma obra;

R2 – Deve ser possível baixar essas configurações para um dispositivo móvel;

R3 – Deve ser possível enviar as informações coletadas para a aplicação web ao fim do dia;

R4 – Deve ser possível registrar a aferição dos dados em um dispositivo portátil de forma off-line;

R5 – Deve ser possível extrair relatórios com os indicadores de produtividade de uma obra.

Um grupo de pessoas da XSOA, que posteriormente faria parte da equipe de desenvolvimento, foi destacado para auxiliar o PO no refinamento dos requisitos iniciais.

10.3.1R1 – Configuração de atividades

Após algumas entrevistas e workshops entre XSOA e PO, verificarem-se mais alguns detalhes sobre o funcionamento esperado da solução.

Haverá uma ou mais pessoas no escritório da SmartBrick, que terão que desempenhar tarefas diversas no fluxo de trabalho da solução. Essas pessoas serão responsáveis por analisar o projeto da obra e identificar que atividades serão passíveis de acompanhamento naquela obra.

Por exemplo, numa obra de construção de um prédio residencial, serão desempenhadas atividades de fundação da obra, construção de paredes, construção de laje para outro andar, lixamento de superfícies, pintura de paredes, pintura de teto, acabamento, colocação de tomadas elétricas, entre outras. Numa obra de construção de um quadra de esportes pública, por outro lado, não será desempenhada, por exemplo, a construção de paredes ou laje para um outro andar, somente de alambrados.

Verificou-se também que essas pessoas que ficam no escritório fazem interface com as construtoras clientes, logo, devem ter acesso completo às informações de qualquer obra do sistema.

Por outro lado, as pessoas que estão na obra fiscalizando seu andamento, devem ter o mínimo de informações possíveis, de forma a evitar erros na entrada de dados, o que prejudicaria o fornecimento de indicadores a um cliente. Por exemplo, num caso onde um fiscal está fisicamente localizado na obra X, verificando o andamento da construção de uma parede, se ele acidentalmente entrar dados relativos ao acabamento de uma parede na obra Y, os indicadores de ambas as obras ficarão comprometidos, gerando prejuízos à construtora cliente.

Assim, verificou-se que cada obra deve ser configurável, de forma que o fiscal só veja no dispositivo móvel os dados pertinentes às obras às quais foi designado. Ou seja, ao baixar as configurações para o dispositivo móvel, o fiscal não deve ter visibilidade de todas as obras e todas as atividades cadastradas no sistema; ele deve ter visibilidade somente das obras às quais foi designado, e respectivas atividades.

Por outro lado, a pessoa que fica no escritório e faz interface com cliente, também pode ter de ir até uma obra e fiscalizar seu andamento. Isso pode ser necessário, por exemplo, no caso de ausência de algum fiscal por motivos de saúde ou caso uma construtora cliente discorde da medição e seja necessária uma intervenção para verificar se a fiscalização está sendo feita da forma correta. Nesse caso, como dito anteriormente, essa pessoa deve ter acesso completo, o que significa dizer que, ao baixar as configurações para o dispositivo, essa pessoa deve poder enxergar todas as obras, com respectivas atividades configuradas.

De posse dessas informações, dado seu o acesso completo ao sistema, convencionou-se o papel de nome “Administrador”, representando a pessoa que faz interface com as obras e tem acesso completo às informações de todas as obras. Analogamente, o papel “Fiscal” foi estabelecido para ilustrar a pessoa que somente acompanha o andamento das obras.

Além disso, verificou-se a existência de alguns requisitos adicionais:

R6 – Deve ser possível cadastrar obras;

R7 – Deve ser possível cadastrar atividades;

R8 – Deve ser possível cadastrar usuários no sistema.

Assim, por fim, chegou-se ao entendimento de que a configuração de atividades engloba os seguintes pontos:

- A configuração de atividades estabelece a ligação entre uma obra (pré-cadastrada no sistema), as diversas atividades (também pré-cadastradas) a serem acompanhadas,

e os fiscais (usuários pré-cadastrados) que farão o acompanhamento daquela obra.

- Somente um usuário com perfil "Administrador" tem acesso aos cadastros (R5, R6, R7) e à configuração de obras. Ao perfil "Fiscal" só é permitido interagir com o dispositivo móvel, por meio das obras a si designadas.
- Um usuário com perfil "Administrador" tem acesso a todas as funcionalidades da aplicação web, e a todas as obras, quando utilizando o aplicativo móvel.

Para melhor entendimento, foi criado o *wireframe* apresentado na Figura 10.1, ilustrando os conceitos relativos a tela de configuração de obras.

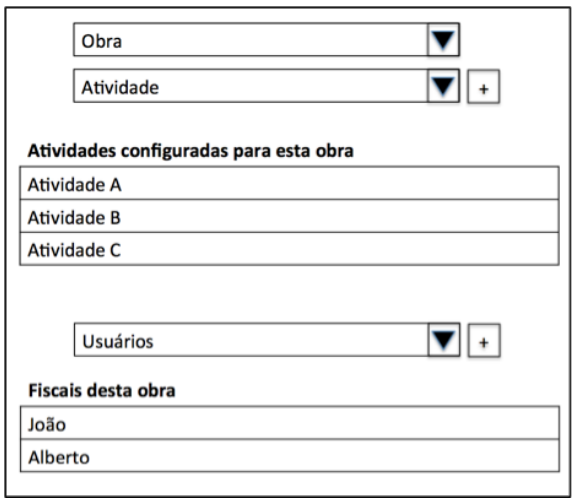


Figura 10.1: *Wireframe* de entendimento do produto esperado

10.3.2R2 – Baixar configurações

Conforme determinado no requisito R1, sobre configuração de atividades, o *download* de informações para o dispositivo móvel deverá levar em consideração o perfil do usuário que está utilizando o dispositivo naquele momento. Se o usuário tiver o perfil "Fiscal", só deverá receber as configurações das obras as quais foi designado. Se seu perfil for "Administrador", deverá receber todas as obras disponíveis no sistema.

Foi verificado também que, por restrições de infra-estrutura e consumo de banda, o tamanho das mensagens transitadas via serviço entre aplicativo móvel e aplicação *web* não deveria exceder 500KB.

10.3.3R3 – Enviar informações

Ao fim do dia, com todos os dados de produtividade coletados e armazenados no dispositivo, o usuário fica responsável por enviar essas informações para a aplicação *web*

via serviço. Esta sincronização é bastante simples, bastando pressionar um botão no dispositivo. O serviço, por sua vez, fica responsável por persistir os dados no sistema para posterior uso pelos relatórios do sistema. Aqui também se aplica a limitação de 500KB para tamanho de mensagens.

10.4Escopo deste exemplo

Os oito requisitos iniciais foram divididos da seguinte forma entre as equipes:

S	W	D
R2, R3	R1, R5, R6, R7, R8	R4

Apesar de terem sido identificados, até o momento, oito requisitos, a partir dessa seção o escopo desse exemplo será limitado aos desdobramentos relativos a equipe S, responsável pela construção de serviços, o qual é o foco desta dissertação e da LPS proposta. Para servir ao propósito de ilustração da proposta, este exemplo se restringirá a exemplificar somente a execução da primeira Sprint.

10.5*User Stories*

A partir do entendimento na necessidade da SmartBrick, conforme descrito acima, as seguintes *User Stories* foram criadas, de forma a suportar o entendimento e o desenvolvimento da solução. Estas foram descritas usando o formato conhecido como Connextra, sugerido por COHN [61].

Assim, os requisitos iniciais **R2** e **R3** foram descritos da seguinte forma:

EU COMO Fiscal
DESEJO baixar as configurações das obras a mim designadas
A FIM DE aferir como a mão de obra disponível em uma obra pode ser melhor aproveitada

EU COMO Administrador
DESEJO baixar as configurações de todas as obras disponíveis no sistema
A FIM DE poder verificar pessoalmente a fiscalização de uma obra caso haja divergência entre o cliente e a SmartBrick

EU COMO Fiscal
DESEJO enviar os dados coletados ao longo do dia
A FIM DE fornecer indicadores de como a mão de obra disponível em uma obra pode ser melhor aproveitada ao cliente da SmartBrick

EU COMO Administrador
DESEJO enviar os dados coletados ao longo do dia
A FIM DE fornecer indicadores de como a mão de obra disponível em uma obra pode ser melhor aproveitada ao cliente da SmartBrick

Note-se que não há relação de restrição entre o número de requisitos iniciais e o número de *User Stories*. Os requisitos podem ser desmembrados ou agrupados em números distintos de *User Stories*.

Note-se também que a cláusula *A FIM DE* identifica não o propósito imediato da ação, mas sim o valor de negócio gerado ao fim.

10.6Critérios de aceitação

Conforme definido por COHN [61], as *User Stories* não definem requisitos em si; são apenas "lembretes", usado para guiar a conversa e entendimento entre as partes. O que efetivamente define o funcionamento esperado de uma funcionalidade são seus critérios de aceitação. Estes servirão de parâmetro para a equipe de desenvolvimento, ilustrando o comportamento esperado pelo PO em relação a solução e os testes que o próprio PO fará quando da entrega do produto.

Assim, os seguintes critérios de aceitação foram elaborados para apoiar o desenvolvimento dos requisitos **R2** e **R3**:

- Verificar se o Fiscal consegue baixar somente as suas obras (passar);
- Verificar se o Fiscal consegue baixar obras assinaladas para outras pessoas (falhar);
- Verificar se o Fiscal consegue baixar todas as obras do sistema (falhar);
- Verificar se cada obra baixada pelo Fiscal contém somente as atividades configuradas para a mesma (passar);

- Verificar se cada obra baixada pelo Administrador contém somente as atividades configuradas para a mesma (passar);
- Verificar se cada obra baixada pelo Fiscal contém todas as atividades cadastradas no sistema (falhar);
- Verificar se cada obra baixada pelo Administrador contém todas as atividades cadastradas no sistema (falhar);

- Verificar se um usuário sem perfil consegue baixar alguma obra (falhar);

- Verificar se os dados da obra X enviados por um Fiscal aparecem somente nos indicadores da obra X (passar);
- Verificar se os dados da obra X enviados por um Fiscal aparecem nos indicadores da obra Y (falhar);
- Verificar se os dados da obra X enviados por um Fiscal aparecem nos indicadores da obra X e também nos indicadores da obra Y (falhar);

- Verificar se os dados da obra X enviados por um Administrador aparecem somente nos indicadores da obra X (passar);
- Verificar se os dados da obra X enviados por um Administrador aparecem nos indicadores da obra Y (falhar);
- Verificar se os dados da obra X enviados por um Administrador aparecem nos indicadores da obra X e também nos indicadores da obra Y (falhar);

- Verificar se um usuário sem perfil consegue enviar os dados das obras (falhar);

- Verificar se uma requisição de download respeita o limite de 500KB (passar);
- Verificar se uma requisição de download excede 500KB (falhar);
- Verificar se uma requisição de upload respeita o limite de 500KB (passar);
- Verificar se uma requisição de upload excede 500KB (falhar);

10.7Preparação

Antes do início da primeira iteração, foi previamente definido que o *backlog* total da equipe S para o projeto seria conforme abaixo:

Prioridade	***User Story***	***Story Points***
1	Baixar configurações para Fiscal	5
2	Enviar dados de Fiscal	5
3	Baixar configurações para Administrador	3
4	Enviar dados de Administrador	2

A pontuação de cada item do *backlog* foi dada pela equipe S, e a ordem inicial foi definida considerando a prioridade dada pelo PO. A equipe também estimou que, inicialmente, sua velocidade seria de 9 pontos por Sprint.

O PO e as equipes definiram que toda comunicação que demandasse registro (decisões, notificações, pontos de esclarecimento) seriam registrados em uma ferramenta wiki.

Ao avaliar a prioridade em relação à velocidade esperada pela equipe, a equipe percebeu que não poderia se comprometer com a entrega de mais do que o primeiro item, uma vez que a pontuação dos dois primeiros itens somados (10) excede a sua capacidade.

Desta forma, para aproveitar melhor a capacidade da equipe, o PO optou por mudar a priorização dos itens, de forma que a equipe entregasse um número maior de itens ao final da Sprint:

Prioridade	***User Story***	***Story Points***
1	Baixar configurações para Fiscal	5
2	Baixar configurações para Administrador	3
3	Enviar dados de Fiscal	5
4	Enviar dados de Administrador	2

Assim, ficou definido que a equipe **S** entregaria, ao final da primeira Sprint, a funcionalidade de download de configurações para o dispositivo portátil, tanto para usuários

com o perfil “Fiscal” quanto aqueles com o perfil “Administrador”.

10.8Instanciação da LPS

Para fins deste exemplo, os seguintes componentes concretos foram escolhidos nos pontos onde há variação:

Componente abstrato	**Componente concreto escolhido**
UNR.FORE.ABST.0005 - Fornecer requisitos da iteração à equipe	UNR.APRE.CONC.0006 - Apresentar requisitos da iteração à equipe
UNR.FOIN.ABST.0008 - Fornecer informações de potencial de reuso à equipe	UNR.APIN.CONC.0009 - Apresentar informações de potencial de reuso à equipe
UNR.COTI.ABST.0025 - Codificar guiado por testes de integração	UNR.COTR.CONC.0028 - Codificar testes de integração entre serviços por meio de invocações reais

10.9Execução da LPS

Componente: UNR.AVRE.CONC.0001 - Avaliar requisitos

Por se tratar da primeira Sprint, as equipes acordaram com o PO de fazer uma validação conjunta dos requisitos iniciais, de forma a identificar de forma antecipada impedimentos globais. Foi agendado um evento de longa duração, combinando as atividades de apresentação e avaliação de requisitos da primeira Sprint de cada equipe.

As equipes **W** e **S** se reuniram fisicamente no mesmo escritório, conectados por ferramenta de videoconferência à equipe **D**.

Uma vez que o PO passaria a maior parte do tempo de projeto mais próximo das equipes **W** e **S**, pelo fato de estarem todos na mesma cidade, a XSOA solicitou que o PO participasse desta primeira validação de requisitos no escritório da equipe **D**, de forma a diminuir sua sensação de distância em relação ao PO e aumentar a coesão do time, o que

foi aceito pelo PO.

Atividade: Verificar consistência geral dos requisitos

Algumas perguntas feitas ao PO pela equipe S foram respondidas ao longo do evento, como, por exemplo:

- **P**: Qual o número médio de atividades em uma obra?
- **R**: 30

- **P**: Há algum nível de hierarquia entre as atividades? as atividades são agrupadas de alguma forma?
- **R**: Sim, as atividades podem ser organizadas de forma hierárquica, com no máximo 2 níveis (raiz e folha, sem atividades intermediárias)

- **P**: Se o usuário adiciona uma atividade raiz à configuração, todos os itens são baixados para o dispositivo? Ou só a atividade raiz?
- **R**: Se uma atividade raiz é adicionada à configuração, todas as atividades folha daquela raiz são baixadas para o dispositivo. De forma geral, o dispositivo só lida com atividades raiz.

Outras perguntas, dos pontos de vista funcional e não-funcional, não puderam ser prontamente respondidas pelo PO:

- **P**: Quantos usuários farão *downloads* ou *uploads* em paralelo?
- **P**: Os dispositivos serão de uso compartilhado ou particular? Caso sejam de uso compartilhado, cada usuário baixará sua configuração quando de seu primeiro uso? Ou o dispositivo deverá armazenar um conjunto de configurações de obras, habilitando a visão de cada configuração de acordo com os usuários configurados?

Os pontos sem resposta foram registrados na wiki do projeto, ficando o PO responsável por providenciar suas respostas. A equipe e o PO concluíram ainda que seria possível prosseguir com a primeira Sprint sem essas respostas.

Ficou acordado e registrado na wiki que seria assumido que não haveria mais do que 30 usuários fazendo *downloads* ou *uploads* simultaneamente. Também foi registrado que seria assumida a premissa de que os dispositivos seriam de uso privado e, caso essa definição mudasse, a funcionalidade seria refatorada.

Atividade: Verificar qualidade das *User Stories*

A equipe **S** avaliou as *User Stories* em relação ao modelo INVEST e não houve necessidade de nenhum esclarecimento ou modificação.

Atividade: Verificar qualidade dos critérios de aceitação

A equipe avaliou os critérios em relação ao modelo SMART e não houve necessidade de nenhum esclarecimento ou modificação.

Atividade: Notificar envolvidos

A equipe registrou na wiki que não houve mais pontos a serem esclarecidos e que os requisitos estão maduros e consistentes.

Componente: UNR.ESRE.CONC.0002 - Esclarecer requisitos

Como a equipe registrou, durante a execução da atividade **Notificar envolvidos**, que não havia pontos a serem esclarecidos e que os requisitos estavam maduros e consistentes, foi decidido que este componente não seria executado.

Componente: UNR.AJGR.CONC.0003 - Ajustar granularidade das User Stories

Como a equipe registrou, durante a execução da atividade **Notificar envolvidos**, que não havia pontos a serem esclarecidos e que os requisitos estavam maduros e consistentes, foi decidido que este componente não seria executado.

Componente: UNR.REES.CONC.0004 - Refinar estimativas

Como não houve mudança no entendimento nem na granularidade dos requisitos, foi decidido que este componente não seria executado.

Componente: UNR.APRE.CONC.0006 - Apresentar requisitos da iteração à equipe

Conforme descrito na execução do componente **UNR.AVRE.CONC.0001 - Avaliar requisitos**, a apresentação dos requisitos foi feita em conjunto com a validação dos mesmos, sem que novas dúvidas ou questionamentos fossem levantados.

Componente: UNR.APIN.CONC.0009 - Apresentar informações de potencial de reuso à equipe

Durante a execução deste componente, a SmartBrick informou que dispõe de um banco de dados com informações de todos os seus funcionários, informações estas expostas por meio de serviços *web*. Desta forma, todos os dados relativos a funcionários da SmartBrick que fossem enviados para o dispositivo teriam que, necessariamente, ser recuperados por meio de uma integração entre o serviço sendo desenvolvido e o serviço exposto por este banco de dados central.

Além disso, a SmartBrick informou que não havia expectativa de reuso do serviço sendo desenvolvido por nenhum outro consumidor. Logo, não haveria necessidade de um *design* complexo, visando flexibilidade desnecessária.

Não foram levantadas novas dúvidas ou questionamentos.

Componente: UNR.REFU.CONC.0012 - Repriorizar funcionalidades

Uma vez que não foram identificadas novas dúvidas ou questionamentos durante a execução dos componentes **UNR.APRE.CONC.0006 - Apresentar requisitos da iteração à equipe** e **UNR.APIN.CONC.0009 - Apresentar informações de potencial de reuso à equipe**, foi decidido que este componente não seria executado.

Componente: UNR.IDME.CONC.0013 - Identificar metáforas

A equipe considerou que os requisitos priorizados para a iteração estavam claros e apresentavam baixa complexidade, não havendo, assim, necessidade de se estabelecer metáforas naquele momento.

Componente: UNR.AUTE.CONC.0014 - Automatizar testes de aceitação

Seguindo as orientações do componente, os critérios de aceitação foram transformados em testes de aceitação funcional e não-funcional automatizados. Foi utilizado o *framework* Cucumber, o qual preconiza o uso de arquivos com a extensão *.feature*.

Os critérios de aceitação funcionais descritos conforme a sintaxe do Cucumber podem ser vistos em `https://github.com/felipecao/xsoa-example/blob/master/src/test/resources/br/unirio/xsoa/endpoint/core/download.feature`. O código executável que automatiza a execução destes critérios como testes pode ser visto em `https://github.com/felipecao/xsoa-example/blob/master/src/test/java/br/unirio/xsoa/endpoint/core/DownloadSteps.java`.

Analogamente, os critérios de aceitação não-funcionais podem ser vistos em `https://github.com/felipecao/xsoa-example/blob/master/src/test/resources/br/unirio/xsoa/endpoint/nonfunctional/nonfunctional.feature`, bem como a sua forma automatizada: `https://github.com/felipecao/xsoa-example/blob/master/src/test/java/br/unirio/xsoa/endpoint/nonfunctional/NonFunctionalSteps.java`.

Componente: UNR.PRCO.CONC.0015 - Projetar contrato mínimo

Atividade: Identificar operações

A partir dos requisitos priorizados para a Sprint, a equipe verificou que seria necessário somente uma operação, para *download* das informações às quais o usuário tem acesso.

Atividade: Identificar entradas e saídas

Seguindo as boas práticas de construção de serviços prescritas por LEE *et al.* [18], esta operação deveria lançar mão de tipos complexos para requisição e resposta. Para o objeto de requisição bastaria saber o usuário que estava fazendo a requisição, ao passo que a resposta deveria fornecer as informações de cada obra a que o usuário tem acesso.

Atividade: Agrupar operações por afinidade

Uma vez que só foi identificada uma operação até o momento, não houve necessidade de agrupá-la com nenhuma outra.

Atividade: Identificar políticas

De acordo com as definições do componente ao qual pertence a atividade, o contrato deve expressar capacidades, restrições e informações semânticas por meio de seu contrato. No entanto, é possível automatizar a publicação de diversas informações.

Para a execução do projeto foi escolhida a linguagem Java, a qual endereça o desenvolvimento de serviços através das especificações JAX-RS e JAX-WS, tendo esta última sido escolhida para a execução do projeto. Esta especificação prevê anotações que automatizam diversas tarefas, entre elas a geração do contrato.

A equipe chegou ao acordo de que informações relativas a políticas de segurança, restrição de tamanho de mensagem e capacidades do serviço seriam publicadas na wiki, uma vez que a equipe tinha pouco conhecimento sobre automatização da publicação destas informações utilizando as anotações do JAX-WS e não dispunha de tempo para aprendê-lo ao longo da Sprint. Foi acordado que esta decisão seria revisitada no futuro, ocasional-

mente resultando na modificação do contrato com estas informações publicadas por meio automático.

Componente: UNR.PRBA.CONC.0016 - Projetar baixo acoplamento

Atividade: Projetar acoplamento da lógica ao contrato

Conforme a descrição da atividade, com os testes de aceitação automatizados criados e o contrato mínimo, este tipo de acoplamento automaticamente já foi tratado, não restando nada a ser feito.

Atividade: Projetar acoplamento do contrato à lógica

De forma análoga à atividade anterior, a criação de testes de aceitação automatizados e a definição do contrato mínimo tornam desnecessária a execução desta atividade, o que se reforça pelo fato de que nenhum artefato pré-existente foi usado para derivar o contrato.

Atividade: Projetar acoplamento do contrato à tecnologia

De forma a prevenir acoplamento inadequado deste tipo, a equipe acordou utilizar tipos de dados simples (por exemplo, array) em detrimento de tipos de dados definidos no Java (por exemplo, *List*) na definição do contrato, atendendo também às boas práticas definidas por LEE *et al.* [18].

Atividade: Projetar acoplamento do contrato à implementação

Para endereçar os requisitos deste componente, a equipe revisitou rapidamente a parte do contrato que seria publicada na wiki para garantir que não continha detalhes de implementação ou plataforma.

Com o mesmo objetivo, foi criado um pequeno serviço de exemplo, utilizando as anotações do JAX-WS, a fim de verificar se o contrato gerado automaticamente conteria algum detalhe da plataforma. Após a publicação deste exemplo, verificou-se que os artefatos gerados pelo JAX-WS não expunham nenhum detalhe da plataforma.

Atividade: Projetar acoplamento do contrato ao escopo funcional

Como este seria o primeiro serviço a integrar o repositório da SmartBrick, verificou-se que não haveria sobreposição de escopo com outros serviços. Além disso, conforme previsto nas informações de potencial de reuso, este serviço seria de uso bastante direcionado, não havendo problemas quanto a medidas de reuso.

Atividade: Projetar conexões físicas

A equipe, em conjunto com o PO, definiu que não havia necessidade de implantação de um mediador de conexões, logo, todas as conexões seriam do tipo ponto-a-ponto. Definiu-se também que esta decisão seria mantida para todos os serviços a serem desenvolvidos ao longo do projeto.

Atividade: Projetar estilo de comunicação

A fim de manter a solução mais simples possível, e considerando o número restrito de usuários esperado, a equipe chegou ao consenso de que uma comunicação síncrona atendia às necessidades da SmartBrick. Definiu-se que esta decisão seria mantida para todos os serviços a serem desenvolvidos ao longo do projeto.

Atividade: Projetar estruturas de dados

Os tipos de dados a serem utilizados no serviço foram derivados a partir das entradas e saídas identificadas nos requisitos. Considerando as boas práticas definidas por LEE *et al.* [18], a equipe decidiu que seriam utilizados tipos de dados simples e que favorecessem a interoperabilidade. Definiu-se que esta decisão seria mantida para todos os serviços a serem desenvolvidos ao longo do projeto.

A representação final dos tipos de dados definidos para a comunicação podem ser vistos em `https://github.com/felipecao/xsoa-example/tree/master/src/main/java/br/unirio/xsoa/endpoint/core`.

Atividade: Projetar tipagem

Uma vez que a verificação de tipagem seria feita de forma automática pelo JAX-WS, sem necessidade de construção de lógicas de validação de tipos, a equipe decidiu dar preferência a tipos de dados fortes, por entender que o uso de tipos fortes daria mais produtividade durante o trabalho, sendo Java uma linguagem fortemente tipada.

Definiu-se que esta decisão seria mantida para todos os serviços a serem desenvolvidos ao longo do projeto.

Atividade: Projetar padrão de interação

Seguindo a recomendação de JOSUTTIS [3], a equipe decidiu priorizar o uso de mensagens auto-contidas ao longo de todo o projeto.

Componente: Projetar abstração

Atividade: Projetar informações de propósito do serviço

Para atender aos requisitos da atividade, a equipe revisitou o contrato não-técnico previamente publicado na wiki e identificou que o mesmo não mencionava o propósito do serviço nem quem o utilizaria. Foi acordado que estas informações seriam adicionadas ao contrato durante o andamento da *Sprint*.

Atividade: Projetar informações de utilização do serviço

De forma análoga à atividade anterior, a equipe acordou que ao longo da *Sprint* adicionaria ao contrato informações sobre SLA e forma de interação. Como as informações de potencial de reuso não indicavam um aumento no número de usuários utilizando o serviço, decidiu-se que não seria necessário implementar nenhuma política de tratamento de carga.

Atividade: Projetar informações de capacidades do serviço

De acordo com o PO, não havia um mapeamento dos processos de negócio da companhia, impedindo que fosse adicionada qualquer informação sobre as capacidades da organização ao contrato.

Atividade: Projetar informações de restrições do serviço

Na execução deste componente, a equipe concordou em, ao longo da *Sprint*, adicionar informações sobre *timeout* de requisições e especificar que não seria usado nenhum protocolo de segurança em particular, bastando informar usuário e senha a cada requisição.

Atividade: Projetar abstração tecnológica

Durante a execução desta atividade, a equipe revisou rapidamente o contrato não-técnico, com o propósito de eliminar quaisquer informações com detalhes tecnológicos, além de ter acordado revisar o contrato não-técnico ao final da Sprint para garantir a aderência a esta atividade.

Atividade: Projetar abstração lógica

De forma análoga à atividade anterior, o contrato foi rapidamente revisado e foi acordado que este seria revisado ao final da *Sprint* de forma a eliminar informações sobre lógica de funcionamento do serviço.

Atividade: Projetar abstração funcional

Para atender a este componente, a equipe acordou manter atenção a operações indevidamente expostas ao longo da construção do serviço em pares.

Componente: Projetar reuso

Atividade: Definir medida de reuso

De acordo com as informações de potencial de reuso fornecidas pelo PO, a medida de reuso do serviço foi classificada como "tático".

Atividade: Projetar conectores

Ainda de acordo com as informações fornecidas pelo PO, o serviço sendo construído deveria reutilizar um serviço interno, que expõe dados de funcionários. Desta forma, a equipe acordou construir um conector que receberia o identificador de uma pessoa, retornando o conjunto de seus dados.

Atividade: Projetar modificações em modelos canônicos

A SmartBrick não possuía ainda um modelo canônico de objetos, o que torna esta atividade dispensável.

Atividade: Notificar consumidores sobre nova versão de contrato

Como esta seria a primeira versão do serviço, não havia consumidores a serem notificados.

Componente: Projetar testes de verificação de isolamento de ambiente

Seguindo a sugestão da atividade, a equipe discutiu rapidamente sobre os requisitos e as lógicas de negócio e chegou ao acordo de que, em princípio, não haveria necessidade de violações de isolamento entre o serviço e o *hardware* onde executaria. A equipe acordou em se manter atenta a este ponto durante a codificação em pares, mas definiu que a suíte de testes executada no ambiente de integração contínua seria suficiente para garantir a aderência à atividade.

Componente: Projetar autonomia

Atividade: Projetar fronteiras funcionais do contrato

Seguindo a sugestão da atividade, como os componentes **UNR.PRCO.CONC.0015** e **UNR.PRBA.CONC.0016** foram executados, a execução desta atividade foi dispensada.

Componente: Projetar ausência de estado

Atividade: Projetar retenção intencional de estado

Como não havia nenhum requisito funcional ou não-funcional indicando a necessidade de retenção de estado, a equipe acordou que era desnecessário executar esta atividade.

Atividade: Projetar testes de carga

Apesar do serviço não reter estado, a equipe concordou que seria importante executar testes de carga com regularidade, de forma a garantir que o desempenho do serviço se manteria dentro dos limites esperados. Assim, a equipe acordou em automatizar um teste que simulasse 30 usuários concorrentes fazendo *download* de suas configurações, onde cada usuário estaria atrelado a 10 obras, número que foi considerado razoável pelo PO.

A equipe também acordou que este teste não deveria ser executado a cada novo código disponível no repositório, de forma a não sobrecarregar o processo de *build*. Desta forma, acordou-se que este teste seria executado duas vezes ao dia: uma no meio do dia, outro a noite.

Atividade: Projetar testes de verificação de idempotência

Conforma a sugestão da atividade, a equipe concordou em acrescentar uma verificação aos testes de carga, verificando se os dados de usuário retornados pelo serviço de fato referiam ao usuário que fez a requisição de *download*.

Componente: Projetar descoberta

Componente: Definir estratégia de evolução de serviço pré-existente

Como não havia nenhum serviço pré-existente, a equipe optou por não executar esta atividade.

Componente: Definir responsáveis pelas tarefas

Atividade: Planejar tarefas

A partir dos pontos discutidos nos componentes de planejamento, a equipe delineou inicialmente as seguintes tarefas:

- Automatizar testes de aceitação;
- Implementar conector com serviço de dados de pessoal;
- Automatizar teste de carga / idempotência;
- Atualizar contrato não-técnico com acordos de projeto;
- Revisar contrato não-técnico;
- Criar tabelas no banco de dados;

- Automatizar criação de tabelas;

Após uma revisão rápida, o primeiro item foi dividido em outros de menor granularidade:

- Escrever teste de aceitação automatizado;
- Implementar camada de persistência;
- Implementar camada de negócio;
- Implementar camada de apresentação;

Atividade: Atribuir responsabilidades

Uma vez definidas as tarefas, os pares de membros da equipe ficaram responsáveis por se responsabilizar por uma tarefa cada. Ao longo da *Sprint*, conforme suas atividades terminassem, cada par de pessoas deveria escolher a próxima tarefa.

Componente: Planejar estratégia de testes

Atividade: Definir formato dos testes de integração

Dado que a SmartBrick não dispõe de um barramento de serviços, e que o serviço de dados de pessoal não dispõe de um ambiente que possa ser usado para execução de testes, a equipe escolheu a abordagem de **testes de integração por meio de simulação de API**, de forma também a não sobrecarregar o serviço de dados de pessoal com testes executados com frequência.

Como já se havia decidido anteriormente pela construção de um conector com o serviço de dados de pessoal, não haveria nenhuma dificuldade adicional, bastaria simular o comportamento do conector.

Atividade: Definir estratégia de codificação de testes

Seguindo as práticas propostas pelo XP, a equipe decidiu que os testes seriam codificados em pares, antes do código de produção, sendo o código de produção gerado como a resposta mais simples aos testes.

Componente: Codificar guiado por testes de integração

De acordo com a estratégia de testes adotada e o modelo de três camadas (persistência, negócio e apresentação) escolhido, só há necessidade de um teste de integração, o qual

diz respeito a lógica de persistência. O código fonte do teste pode ser visto em `https://github.com/felipecao/xsoa-example/blob/master/src/test/java/br/unirio/xsoa/dao/core/SiteDaoTest.java`, bem como o código de produção gerado em resposta a este: `https://github.com/felipecao/xsoa-example/blob/master/src/main/java/br/unirio/xsoa/dao/core/SiteDao.java`.

Componente: Codificar *mocks* de API para teste de integração entre serviços

Nesta estratégia, a ligação entre o serviço de *download* e o serviço de dados de pessoal é simulada na escrita do teste da camada de negócios. Assim, a codificação do *mock* será feita no componente seguinte, durante a codificação dos testes unitários da camada de negócio.

Componente: Codificar guiado por testes unitários

Há duas camadas que podem ser codificadas por testes unitários: a camada de negócio e a camada de apresentação.

No caso da camada de apresentação, o teste deverá injetar dois *mocks* na camada de negócio: um *mock* simulando o conector com o serviço de dados de pessoal, e um *mock* simulando a camada de persistência.

O teste pode ser visto em `https://github.com/felipecao/xsoa-example/blob/master/src/test/java/br/unirio/xsoa/service/core/SiteServiceTest.java`, bem como o código de produção correspondente: `https://github.com/felipecao/xsoa-example/blob/master/src/main/java/br/unirio/xsoa/service/core/SiteService.java`.

Com a codificação tendo sido feita em par, os componentes **UNR.RECO.CONC.0031** e **UNR.ADPS.CONC.0032** foram dispensados. Uma vez que todas as tarefas planejadas tenham sido executadas, a codificação do serviço se dá por terminada, e o próximo componente é executado.

Componente: Disponibilizar serviço por meio automático

Atividades: Publicar contrato técnico de forma automática / Instalar serviço de forma automática

Conforme o definido no componente, ao final da codificação a equipe usou sua ferramenta de integração contínua para instalar o serviço de forma automática. Além disso, conforme definido anteriormente, o contrato técnico seria gerado automaticamente pelo *framework* JAX-WS, bastando a instalação do serviço para que contrato técnico estivesse

automaticamente disponível.

Atividade: Publicar contrato não-técnico de forma automática

A equipe também implementou um *script* utilizando o *framework* Selenium [1], o qual copiava o conteúdo do contrato não-técnico disponível na wiki do projeto e o disponibilizava num *web site* acessível a todas as equipes envolvidas no projeto.

Componente: Executar testes integrados manuais

Conforme sugerido no componente, um par de pessoas ficou responsável por recuperar os critérios de aceitação previamente definidos e executá-los manualmente contra o serviço instalado. Ao fim da execução, não foram encontrados defeitos em relação ao ponto de vista funcional.

Componente: Executar testes exploratórios

Conforme a descrição do componente, um par de pessoas se responsabilizou por testar variações dos critérios de aceitação que não foram previstas, bem como outras situações limítrofes. Ao final da execução, não foram encontrados defeitos.

Componente: Demonstrar produto da iteração

Ao final da *Sprint*, a equipe **S** organizou a demonstração do produto da iteração. Os membros de outras equipes não foram convidados, uma vez que só haveria interesse para os mesmos se tivesse havido alguma mudança em relação ao planejado, o que não foi o caso. A equipe executou manualmente alguns testes funcionais e o PO aceitou a entrega, sem apontar nenhum erro e sem necessidade de ajustes.

Componente: Realizar Mega Planning

Como ao início da *Sprint* optou-se por um evento discutindo os requisitos de todas as equipes em conjunto, este componente foi dispensado na primeira iteração.

Componente: Realizar Stand-Up Meeting

Diariamente, os membros de cada equipe do projeto ficaram responsáveis por apontar as tarefas concluídas no dia anterior, as tarefas em que estavam trabalhando e possíveis impedimentos que houvessem identificado. Caso fosse identificado algum impedimento, um membro da equipe ficava responsável por discutí-lo com o PO.

Componente: Realizar Mega Stand-Up

[1] `http://www.seleniumhq.org/`

Na metade da primeira *Sprint*, representantes de cada equipe se reuniram para falar sobre os itens que foram planejados para suas *Sprints*, o andamento de suas atividades e impedimentos identificados. Caso fosse identificado um impedimento compartilhado, os representantes das equipes impactadas ficavam responsáveis por discuti-lo com o PO.