



TÓPICOS ESPECIAIS



Professora Dra. Elisa Hatsue Moriya Huzita
Professora Esp. Janaína Aparecida de Freitas

UNICESUMAR

Av. Guedner, 1610 - Jardim Aclimação
Cep 87050-900 - MARINGÁ - PARANÁ
unicesumar.edu.br
44 3027.6360

UNICESUMAR EDUCAÇÃO A DISTÂNCIA

NEAD - Núcleo de Educação a Distância
Bloco 4 - MARINGÁ - PARANÁ
unicesumar.edu.br
0800 600 6360

as imagens utilizadas neste
livro foram obtidas a partir
do site SHUTTERSTOCK.COM

FICHA CATALOGRÁFICA

C397 **CENTRO UNIVERSITÁRIO DE MARINGÁ.** Núcleo de Educação a Distância; **HUZITA**, Elisa Hatsue Moriya; **FREITAS**, Janaína Aparecida de.

Tópicos Especiais. Elisa Hatsue Moriya Huzita; Janaína Aparecida de Freitas.
Maringá-Pr.: UniCesumar, 2019.
232 p.
"Graduação - EaD".

1. Tópicos especiais. 2. Engenharia de Software. 3. EaD. I. Título.

ISBN 978-85-459-1955-1

CDD - 22 ed. 005.5
CIP - NBR 12899 - AACR/2

Ficha catalográfica elaborada pelo bibliotecário
João Vivaldo de Souza - CRB-8 - 6828

Impresso por:

Reitor

Wilson de Matos Silva

Vice-Reitor

Wilson de Matos Silva Filho

Pró-Reitor Executivo de EAD

William Victor Kendrick de Matos Silva

Pró-Reitor de Ensino de EAD

Janes Fidélis Tomelin

Presidente da Mantenedora

Cláudio Ferdinandi

NEAD - Núcleo de Educação a Distância**Diretoria Executiva**

Chrystiano Mincoff

James Prestes

Tiago Stachon

Diretoria de Graduação e Pós-graduação

Kátia Coelho

Diretoria de Permanência

Leonardo Spaine

Diretoria de Design Educacional

Débora Leite

Head de Produção de Conteúdos

Celso Luiz Braga de Souza Filho

Head de Curadoria e Inovação

Tania Cristiane Yoshie Fukushima

Gerência de Produção de Conteúdo

Diogo Ribeiro Garcia

Gerência de Projetos Especiais

Daniel Fuverki Hey

Gerência de Processos Acadêmicos

Taessa Penha Shiraishi Vieira

Gerência de Curadoria

Carolina Abdalla Normann de Freitas

Supervisão de Produção de Conteúdo

Nádila Toledo

Coordenador de Conteúdo

Fabiana de Lima

Designer Educacional

Janaína de Souza Pontes

Projeto Gráfico

Jaime de Marchi Junior

José Jhony Coelho

Arte Capa

Arthur Cantareli Silva

Ilustração Capa

Bruno Pardinho

Editoração

Victor Augusto Thomazini

Qualidade Textual

Felipe Veiga da Fonseca

Ilustração

Marcelo Yukio Goto

Marta Sayuri Kakitani

Bruno de Camargo Pinhata



Professor
Wilson de Matos Silva
Reitor

Em um mundo global e dinâmico, nós trabalhamos com princípios éticos e profissionalismo, não só para oferecer uma educação de qualidade, mas, acima de tudo, para gerar uma conversão integral das pessoas ao conhecimento. Baseamo-nos em 4 pilares: intelectual, profissional, emocional e espiritual.

Iniciamos a Unicesumar em 1990, com dois cursos de graduação e 180 alunos. Hoje, temos mais de 100 mil estudantes espalhados em todo o Brasil: nos quatro campi presenciais (Maringá, Curitiba, Ponta Grossa e Londrina) e em mais de 300 polos EAD no país, com dezenas de cursos de graduação e pós-graduação. Produzimos e revisamos 500 livros e distribuímos mais de 500 mil exemplares por ano. Somos reconhecidos pelo MEC como uma instituição de excelência, com IGC 4 em 7 anos consecutivos. Estamos entre os 10 maiores grupos educacionais do Brasil.

A rapidez do mundo moderno exige dos educadores soluções inteligentes para as necessidades de todos. Para continuar relevante, a instituição de educação precisa ter pelo menos três virtudes: inovação, coragem e compromisso com a qualidade. Por isso, desenvolvemos, para os cursos de Engenharia, metodologias ativas, as quais visam reunir o melhor do ensino presencial e a distância.

Tudo isso para honrarmos a nossa missão que é promover a educação de qualidade nas diferentes áreas do conhecimento, formando profissionais cidadãos que contribuam para o desenvolvimento de uma sociedade justa e solidária.

Vamos juntos!



Janes Fidélis Tomelin

Pró-Reitor de Ensino de EaD

Kátia Solange Coelho

Diretoria de Graduação e Pós

Débora do Nascimento Leite

Diretoria de Design Educacional

Leonardo Spaine

Diretoria de Permanência

Seja bem-vindo(a), caro(a) acadêmico(a)! Você está iniciando um processo de transformação, pois quando investimos em nossa formação, seja ela pessoal ou profissional, nos transformamos e, consequentemente, transformamos também a sociedade na qual estamos inseridos. De que forma o fazemos? Criando oportunidades e/ou estabelecendo mudanças capazes de alcançar um nível de desenvolvimento compatível com os desafios que surgem no mundo contemporâneo.

O Centro Universitário Cesumar mediante o Núcleo de Educação a Distância, o(a) acompanhará durante todo este processo, pois conforme Freire (1996): “Os homens se educam juntos, na transformação do mundo”.

Os materiais produzidos oferecem linguagem dialógica e encontram-se integrados à proposta pedagógica, contribuindo no processo educacional, complementando sua formação profissional, desenvolvendo competências e habilidades, e aplicando conceitos teóricos em situação de realidade, de maneira a inseri-lo no mercado de trabalho. Ou seja, estes materiais têm como principal objetivo “provocar uma aproximação entre você e o conteúdo”, desta forma possibilita o desenvolvimento da autonomia em busca dos conhecimentos necessários para a sua formação pessoal e profissional.

Portanto, nossa distância nesse processo de crescimento e construção do conhecimento deve ser apenas geográfica. Utilize os diversos recursos pedagógicos que o Centro Universitário Cesumar lhe possibilita. Ou seja, acesse regularmente o Studeo, que é o seu Ambiente Virtual de Aprendizagem, interaja nos fóruns e enquetes, assista às aulas ao vivo e participe das discussões. Além disso, lembre-se que existe uma equipe de professores e tutores que se encontra disponível para sanar suas dúvidas e auxiliá-lo(a) em seu processo de aprendizagem, possibilitando-lhe trilhar com tranquilidade e segurança sua trajetória acadêmica.

Professora Dra. Elisa Hatsue Moriya Huzita

Possui doutorado em Engenharia Elétrica pela Universidade de São Paulo (USP/1995), mestrado em Ciência da Computação pela Universidade de São Paulo (USP/1985) e graduação em Matemática pela Universidade Estadual Paulista Júlio de Mesquita Filho (UNESP/1978). Atualmente é professora titular na Universidade Estadual de Maringá. Tem experiência na área de Ciência da Computação, com ênfase em Engenharia de Software, atuando principalmente nos seguintes temas: ambientes de desenvolvimento de software, software distribuído, desenvolvimento distribuído de software, modelagem de contexto, aplicações sensíveis a contexto, desenvolvimento colaborativo de software e disen.

Para informações mais detalhadas sobre sua atuação profissional, pesquisas e publicações, acesse seu currículo disponível no seguinte endereço: <<http://lattes.cnpq.br/6545213253559397>>.

Professora Esp. Janaína Aparecida de Freitas

Especialização MBA em Teste de Software pela Universidade Ceuma (UNICEUMA/2012). Graduação em Informática pela Universidade Estadual de Maringá (UEM/2010). Atualmente, cursa Licenciatura em Letras - Português/ Inglês no Centro Universitário Cesumar (UniCesumar). Trabalhou na iniciativa privada, na área de Análise de Sistemas e Testes de Software. Tem experiência na área de Engenharia de Software com ênfase em Análise de Requisitos, Gestão de Projetos de Software, Métricas e Estimativas, Qualidade e Teste de Software. É professora mediadora dos cursos de graduação Análise e Desenvolvimento de Sistemas (ADS) e Sistemas para Internet (SI) na modalidade de Ensino a Distância (EAD) no Unicesumar.

Para informações mais detalhadas sobre sua atuação profissional, pesquisas e publicações, acesse seu currículo disponível no seguinte endereço: <<http://lattes.cnpq.br/4906244382612830>>.

APRESENTAÇÃO

TÓPICOS ESPECIAIS

SEJA BEM-VINDO(A)!

Prezado(a) aluno(a), seja bem-vindo(a) à disciplina de Tópicos Especiais. Nesta disciplina, iremos abordar vários conteúdos sobre o problema de entregar software, o desenvolvimento de Sistemas Críticos, a Reengenharia e a Engenharia Reversa. Falaremos também sobre a reutilização de software, a refatoração para padrões e as tendências emergentes da engenharia de software.

As unidades do livro foram organizadas de forma que estejam vinculadas, ou seja, que a unidade seguinte sempre esteja vinculada com a unidade anterior. Portanto, é bom que você leia e entenda todo o conteúdo de uma unidade para passar para a próxima.

Vamos iniciar vendo na unidade I uma apresentação sobre os princípios da Entrega Contínua de Software, os problemas da entrega e quais as práticas necessárias para realizá-la. Ainda vamos descrever os pré-requisitos para entendermos o pipeline de implantação e de alguns antipadrões que muitas equipes de desenvolvimento de software tem. Falaremos também sobre o Gerenciamento de Configuração de Software (SCM – Software Configuration Management) ou GCS que é uma atividade de apoio destinada a gerenciar as mudanças e alguns conceitos básicos sobre a Integração contínua e seus princípios.

Seguindo para a unidade II, vamos apresentar uma visão geral sobre os Sistemas Críticos, Engenharia de Segurança, Reengenharia e Manutenção e Engenharia Reversa. A confiança é a propriedade mais importante em um sistema crítico. Abordaremos sobre os conceitos de um Sistema Sociotécnico – que inclui pessoas, software e hardware – mostrando o que é necessário para ter uma perspectiva de proteção e uma confiança no software. Falaremos sobre falhas, confiança e a proteção de software e como elas não devem ser tratadas isoladamente e como elas podem afetar drasticamente um sistema.

Depois, seguindo para a unidade III, vamos conhecer os principais conceitos sobre o Reuso de Software e suas vantagens e desvantagens. Seguindo, vamos falar sobre o Reuso de Componentes e Frameworks e a Engenharia de Software baseada em Componentes.

Na unidade IV, passaremos a introduzir conceitos e princípios sobre a refatoração para padrões e como ela pode ser aplicada de maneira controlada e eficiente, que não introduza erros no código e que melhore metódicamente a sua estrutura.

E por fim, na unidade V, vamos aprender sobre as Tendências Emergentes da Engenharia de Software e como essas tendências têm efeito sobre a tecnologia de engenharia de software, seus cenários de negócios, organizacionais, mercado e cultural. Vamos falar também sobre Sistemas Distribuídos e arquiteturas orientadas a serviços (SOA).

Espero que sua leitura seja agradável e que esse conteúdo possa contribuir para seu crescimento pessoal e profissional.

Assim, convido você, caro(a) aluno(a), a entrar nessa jornada com empenho, dedicação e muita sede por conhecimento! Vamos começar nossos estudos?

Boa leitura!

SUMÁRIO

UNIDADE I

COMO ENTREGAR SOFTWARE

15	Introdução
16	O Problema de Entregar Software
23	Princípios da Entrega de Software
27	Gerenciamento de Configuração de Software
40	Integração Contínua
46	Considerações Finais
53	Referências
54	Gabarito

UNIDADE II

DESENVOLVIMENTO DE SISTEMAS

57	Introdução
58	Sistemas Críticos
69	Engenharia de Segurança
75	Reengenharia e Manutenção de Software
85	Engenharia Reversa
92	Considerações Finais
101	Referências
102	Gabarito



SUMÁRIO

UNIDADE III

REUTILIZAÇÃO DE SOFTWARE

-
- 105 Introdução
 - 106 Reutilização de Software
 - 108 Vantagens e Desvantagens da Reutilização de Software
 - 110 Reutilização de Componentes, Padrões e Frameworks
 - 124 Engenharia de Software Baseada em Componentes
 - 129 A Evolução da Reutilização de Software
 - 132 Considerações Finais
 - 139 Referências
 - 142 Gabarito

UNIDADE IV

REFATORAÇÃO PARA PADRÕES

-
- 145 Introdução
 - 146 Refatoração
 - 155 Catálogo de Refatoração para Padrões
 - 169 O Catálogo de Padrões de Projeto
 - 173 Considerações Finais
 - 180 Referências
 - 181 Gabarito



SUMÁRIO

UNIDADE V

TENDÊNCIAS EMERGENTES DA ENGENHARIA DE SOFTWARE

185 Introdução

186 Sistemas Distribuídos

198 Arquitetura Orientada a Serviço (SOA – Service Oriented Architecture)

207 Tendências Leves

218 Considerações Finais

228 Referências

229 Gabarito

230 CONCLUSÃO



COMO ENTREGAR SOFTWARE

Objetivos de Aprendizagem

- Entender o problema de Entrega do Software.
- Compreender os princípios que envolvem a Entrega de Software.
- Estudar os conceitos básicos do Gerenciamento de Configuração de Software.
- Definir os conceitos da Integração Contínua de Software.

Plano de Estudo

A seguir, apresentam-se os tópicos que você estudará nesta unidade:

- O Problema de Entregar Software
- Princípios da Entrega de Software
- Gerenciamento de Configuração de Software
- Integração Contínua

INTRODUÇÃO

Olá, aluno(a)! Esta unidade tem por objetivo apresentar os princípios da Entrega Contínua de Software, os problemas da entrega e quais as práticas necessárias para realizá-la.

Vamos começar compreendendo o problema de entregar software. E para isso, temos que pensar na seguinte questão: como otimizar a entrega de software sem impactar no orçamento e na qualidade, com prazos cada vez menores? Novas funcionalidades são adicionadas ou as que já existem são melhoradas e os defeitos encontrados são corrigidos e, a partir disso, surge um problema que muitos profissionais da área de desenvolvimento de software estão enfrentando: como entregar um sistema confiável aos usuários, de forma rápida, com poucos riscos e com qualidade? Para responder a estas perguntas, temos que conhecer o conceito de Entrega Contínua (*Continuous Delivery*) que é o processo de implantação contínua em ambiente de produção. Seu objetivo é encontrar maneiras de entregar software com valor e com qualidade de forma eficiente, rápida e confiável.

Ainda vamos descrever os pré-requisitos para entendermos o *pipeline* de implantação e de alguns antipadrões que muitas equipes de desenvolvimento de software possuem. Também veremos que a entrega de software possui alguns princípios que devem ser seguidos para que o processo de entrega da versão seja eficaz.

Vamos descrever sobre o Gerenciamento de Configuração de Software (SCM – *Software Configuration Management*) ou GCS que é uma atividade de apoio destinada a gerenciar as mudanças, identificando artefatos que precisam ser alterados, as relações entre eles, controle de versão destes artefatos, controlando estas mudanças e auditando e relatando todas as alterações feitas no software.

Também conheceremos alguns conceitos básicos sobre a Integração Contínua e seus princípios, pois a ideia principal da integração contínua é a diminuição dos problemas e riscos por meio de um melhor monitoramento das mudanças e da integração frequente do código.

Preparado(a) para começar? Então, vamos seguir em frente. Boa leitura e bons estudos.



O PROBLEMA DE ENTREGAR SOFTWARE

Com o aumento e disseminação de conceitos como: Big Data, Cloud Computing e Internet das Coisas (IoT), passamos a ter uma interação em tempo real por meio da Internet, jamais vistos. Com isso, surgem as mudanças constantes de cenários, de requisitos e códigos e as exigências de entrega de versões rápidas e desenvolvimento de *releases* em prazos cada vez mais curtos.

Entretanto como otimizar a entrega de software sem impactar no orçamento e na qualidade com prazos cada vez menores? Durante o ciclo de vida de um software novas funcionalidades são adicionadas ou as que já existem são melhoradas e os defeitos encontrados são corrigidos. Com isso, surge um problema que muitos profissionais da área de desenvolvimento de software estão enfrentando: como entregar um sistema confiável aos usuários, de forma rápida, com poucos riscos e com qualidade? Para responder a pergunta, vamos primeiro conhecer os conceitos que envolvem a entrega de software.

A prática de entrega de software é chamada de Entrega Contínua, e ela é uma disciplina de desenvolvimento de software, em que você cria e implanta o software, que pode ser liberado para produção a qualquer momento, ou seja, Entrega Contínua (*Continuous Delivery*) é o processo de implantação contínua em ambiente de produção. Para que isso aconteça, é preciso integrar continuamente todas as mudanças do software que está sendo desenvolvimento, como a criação, o teste e implantação, e que os executáveis estejam prontos para liberar para a produção.

A Entrega Contínua é ativada por uma ferramenta automatizada quando um novo *build* (versão compilada de um software) for publicado com sucesso no ambiente de homologação. Na entrega contínua não significa que o ambiente de produção é modificado a todo momento, mas que o ambiente de produção pode ser alterado se um novo build estiver disponível e se for aprovado para ser liberado para a produção.

A Entrega Contínua tem como objetivo encontrar maneiras de entregar software com valor e com qualidade de forma eficiente, rápida e confiável. Conforme questionam Humble e Farley (2014, p. 3), “o que acontece depois da identificação de requisitos, projetos, desenvolvimento e teste de soluções? Como fazer desenvolvedores, testadores e pessoal de operação trabalharem juntos de maneira eficiente?” Tudo isso depende de um padrão eficaz, que vai desde o desenvolvimento do software até a sua entrega final a produção. O padrão que vamos adotar é chamado de *Pipeline de implantação*, que segundo Humble e Farley (2014, p. 4) é:

[...] em essência, uma implementação automatizada do processo de compilar todas as partes de uma aplicação, implantá-la em um ambiente qualquer – seja de homologação ou produção – testá-la e efetuar sua entrega final. Cada organização tem implementações diferentes de seu pipeline de implantação, dependendo de sua cadeia de valor para entregar software, mas os princípios que regem o pipeline são os mesmos.

Para entendermos o pipeline de implantação, temos um exemplo mostrado na figura 1:



Figura 1 – O pipeline de Implantação
Fonte: Humble e Farley (2014, p. 4).

O pipeline de implantação funciona resumidamente do seguinte modo:

- Cada mudança feita na configuração, no código fonte, no ambiente ou em dados, cria-se uma nova instância do pipeline.
- O restante do pipeline executa uma série de testes para provar que é possível gerar uma entrega de versão.
- Cada teste na versão candidata (*release candidate*) passa a aumentar a confiança.
- Se a versão candidata passa em todos os testes, pode ser realizada a entrega de versão.

O pipeline de implantação tem três objetivos segundo Humble e Farley (2014):

1. Torna cada parte do processo de compilação, implantação, teste e entrega de versão visível a todos os envolvidos, promovendo a colaboração das equipes.
2. Melhora o feedback do processo, identificando e resolvendo os problemas o mais cedo possível.
3. Permite que as equipes entreguem e implantem qualquer versão do software em qualquer ambiente, a qualquer momento por meio de um processo completamente automatizado.

Para Humble e Farley (2014), o pipeline de implantação baseia-se no processo de *Integração Contínua*. Assim, não pode haver entrega contínua sem integração contínua. O processo de compilar o código em ambiente limpo, executar testes e outros processos de qualidade e gerar um *build*, disparado por qualquer modificação no código fonte chamamos de Integração Contínua (*Continuous Integration*). O processo de promover o *build* gerado no processo de integração contínua para ambientes intermediários ou para homologação chamamos de Implantação Contínua (*Continuos Deployment*). O processo de implantação contínua que busca promover os builds para o ambiente de produção chamamos de Entrega Contínua (*Continuous Delivery*), como vimos anteriormente.

SAIBA MAIS



A versão Candidata

O que é uma versão candidata? Uma mudança feita no código pode ou não fazer parte de uma versão. Se você olhar para a mudança e perguntar se deve ou não implantá-la em uma versão, qualquer resposta será apenas um palpite. Todo o processo de compilação, implantação e testes que aplicamos à mudança valida se ela pode ou não fazer parte de uma versão. O processo nos dá confiança de que a mudança é segura. Se o produto resultante é livre de defeitos e segue os critérios de aceitação estabelecidos pelo cliente, então ele pode ser entregue. Cada mudança é, essencialmente, uma versão candidata. Toda vez que uma mudança é introduzida no sistema de versão, a expectativa é que todos os testes resultem em sucesso, produzindo código funcional que pode ser implantado em produção.

Fonte: Humble e Farley (2014, p. 25)

ANTIPADRÕES COMUNS DE ENTREGA DE VERSÃO

O dia da entrega de uma nova versão do sistema tende a ser tenso para as empresas desenvolvedoras de softwares. E por que será que isso acontece? Segundo Humble e Farley (2014, p. 04) “na maioria dos projetos, isso ocorre em função do risco associado ao processo – que transforma cada entrega em algo assustador”. Muitas coisas podem dar errado durante a entrega, se cada passo do processo não for executado perfeitamente. Para evitar esses riscos, precisamos conhecer quais tipos de falhas no processo podem ser evitados. Para isso, vamos a uma lista de antipadrões comuns que podem impedir um processo confiável de entrega de versão.



Tabela 1 - Lista de antipadrões comuns de Entrega de Versão

ANTIPADRÓES	
Implantar software manualmente	<p>Entrega manual é o processo de implantação em que cada passo é visto como separado e atômico, executado individualmente ou por uma equipe. Sinais de antipadrões:</p> <ul style="list-style-type: none"> • Documentação extensa. • Dependência de testes manuais para confirmar que a aplicação está funcionando. • Chamadas frequentes aos desenvolvedores para explicar que algo está errado. • Correções frequentes no processo de entrega. • Ambientes com problemas de configuração diferente (servidores, banco de dados, arquivos, interfaces diferentes etc.). • Entregas de versão que levam mais tempo para executar. • Entrega de versões imprevisíveis.
Implantar em um ambiente similar ao de produção somente quando o desenvolvimento estiver completo	<ul style="list-style-type: none"> • Se foram envolvidos testadores até o momento, eles testaram somente em máquinas de desenvolvimento. • A equipe de operação (time que é usado para a implantação do software em ambiente de homologação e produção) se envolve com a versão do software, no dia da implantação em ambiente de homologação ou no dia da implantação no cliente. • Ambiente similar ao da produção é muito caro e seu acesso é estritamente controlado ou ninguém acha que ele é necessário. • A equipe de desenvolvimento cria os instaladores corretos, arquivo e configuração, as migrações de banco de dados e a documentação necessária para a equipe de operação, mas tudo isso não foi testado em um ambiente de homologação ou produção. • Há pouca ou nenhuma colaboração entre o time de desenvolvimento e a equipe de operação.
Gerência de Configuração manual dos ambientes de produção	<ul style="list-style-type: none"> • Depois de o software ter sido implantado com sucesso várias vezes em um ambiente de homologação, a implantação em produção falha. • Máquinas se comportam de maneiras diferentes – por exemplo, uma suporta menos carga ou demora mais para processar requisições do que outras. • A equipe de operação demora muito para preparar um ambiente para a entrega de uma versão. • A configuração de um sistema é feita por meio da modificação direta da configuração nos sistemas de produção.

Fonte: adaptado de Humble e Farley (2014, p. 5)

Alguns pontos que devemos considerar em relação aos antipadrões, entre ele:

- Com o tempo, o processo de implantação de um software deve tender à automação completa. Pois, conforme Humble e Farley (2014, p. 6), “entregar software que é empacotado com instaladores deve envolver um único processo automatizado que cria o instalador”. Pois, quando o processo não é automático, ocorrerão erros toda vez que ele for executado, mesmo tendo passado por testes de implantação, é difícil rastrear os defeitos.
- Um processo manual precisa ser documentado e isso exige tempo e colaboração da equipe, pois é uma tarefa complexa. Mas lembrando de que a documentação deve estar sempre atualizada e completa, ou a implantação não funcionará.
- O mesmo script de implantação deve ser usado em todos os ambientes de implantação. Assim, os problemas que surgirem na entrega de versão com certeza serão resultados específicos de configuração do ambiente e não dos seus scripts.
- Quanto maior for à diferença entre os ambientes de desenvolvimento e o de produção, menos realistas serão as suposições feitas e é provável que surjam surpresas. A empresa deve ser capaz de recriar o ambiente de produção de forma exata e de maneira automatizada, e a virtualização pode ajudar.
- A equipe de produção deve saber exatamente o que está em produção ou o que vai ser implantado na produção. Assim, qualquer mudança feita, deve ser registrada e auditável. Algumas implantações falham, pois alguém aplicou mudanças e isso não foi registrado.
- O ideal é integrar todas as atividades de teste, implantação e entrega de versão ao processo de desenvolvimento do software, de forma contínua. Segundo Humble e Farley (2014, p. 9), “garanta que todos os envolvidos no processo de entrega de software, de desenvolvedores a testadores, passando por times de implantação e operação, trabalhem juntos desde o começo do projeto”.

Sobre o processo de entrega de um software, Humble e Farley (2014, p. 11) descrevem que:

[...] pode ser estimulante, mas também pode ser exaustivo e deprimente. Quase toda entrega envolve mudanças de última hora, como resolver problemas de conexão ao banco de dados, atualizar a URL de um serviço externo, e assim por diante. Deve haver um modo de introduzir tais mudanças de modo que sejam registradas e testadas. Mais uma vez, a automação é essencial. As mudanças devem ser feitas em um sistema de versionamento e propagadas para produção por meio de processos automatizados. Da mesma forma, deve ser possível usar o mesmo processo para reverter uma versão em produção se a implantar falhar.

As entregas de versão de software devem ser um processo de baixo risco, barato, rápido, frequente e principalmente previsível. Todavia sempre as entregas de versão são assim. Entretanto o que podemos fazer para melhorar? Segundo Humble e Farley (2014), usando pipelines de implantação, combinados com um alto grau de automação de testes e de entrega e com o uso adequado da gerência de configuração, podemos realizar entregas apertando apenas um botão – em qualquer ambiente necessário, seja de testes, desenvolvimento ou produção.

Como alcançar os nossos objetivos? Como todo profissional de desenvolvimento, temos o objetivo de entregar software útil e funcional aos nossos usuários o mais rápido possível. Isso significa que temos que encontrar formas de reduzir o tempo de ciclo entre uma decisão de fazer uma mudança em um software (uma correção ou a inclusão de uma nova funcionalidade) e o momento em que ele será entregue aos usuários com alto valor e alta qualidade de maneira eficiente, confiável e rápida. Para isso, precisamos entregar versões frequentes e automatizadas do software.

- **Automatizadas:** se a entrega não for automatizada, não é possível de repetição. Como os passos são manuais, erros podem aparecer e não existe uma forma de ver o que realmente foi feito.
- **Frequentes:** entregas de versão frequentes, significa que a variação entre elas é menor e isso reduz os riscos e conduzem a um feedback mais rápido.

Nas entregas de versão frequentes e automatizadas, é essencial o feedback. Temos alguns critérios para o feedback (HUMBLE, 2014):

- **Cada mudança deve disparar um processo de feedback:** podemos dividir uma aplicação de software em: código executável, configuração, ambiente de hospedagem e dados. Se cada um deles mudar, o comportamento também poderá mudar. Devemos manter eles sempre sobre controle e garantir que cada mudança seja verificada.
- **O feedback deve ser passado o mais rápido possível:** para o feedback rápido a chave é a automação.
- **A equipe responsável pela entrega da versão deve receber o feedback e aproveitá-lo:** a equipe de envolvidos na entrega de software deve estar igualmente envolvida no processo de feedback. Pois quando acontece alguma coisa, é responsabilidade da equipe em decidir a melhor forma de corrigir.

Para Humble e Farley (2014, p. 24), “em software, quando algo é difícil, a maneira de reduzir o sofrimento é fazer isso com maior frequência, e não com menor”. Ou seja, em vez de integrar ocasionalmente, devemos procurar integrar com frequência como resultado para qualquer mudança feita no sistema. Quando seguimos essa prática, o software está sempre em estado funcional.

PRINCÍPIOS DA ENTREGA DE SOFTWARE

A Entrega Contínua ajuda o software a responder de forma rápida às expectativas dos clientes, aumentando a qualidade dos seus produtos a um baixo custo. Assim, podemos dizer que a entrega contínua é uma prática de entrega de software que exige a criação e implementação do aplicativo e que este pode ser liberado para produção a qualquer instante. E para isso é necessário integrar continuamente as alterações do software em desenvolvimento. Assim como: criar, testar e implantar as versões e estar pronto para liberar para produção.

A entrega de software possui princípios que devem ser seguidos para que o processo de entrega da versão seja eficaz. Os princípios de entrega de software, segundo Humble e Farley (2014) são:

1. Criar um processo de confiabilidade e repetitividade de entrega de versão:

A entrega de uma versão de software deve ser fácil, já que ela foi testada. A repetitividade e a confiabilidade têm como base: automatizar quase tudo e manter tudo o que precisar para compilar, configurar, implantar e testar dentro de um sistema de controle de versão.

Quando implantamos um software temos:

- Um ambiente em que o software será executado (hardware, software, infra-estrutura e sistemas externos) que devem ser fornecidos e gerenciados.
- Neste ambiente o software deve ser instalado com a versão correta.
- Deve ser configurado o software, incluindo dados ou informações conforme a aplicação exige.

Para Humble e Farley (2014, p. 25) “deve-se fazer a implantação da aplicação em produção usando um processo completamente automatizado a partir do sistema de controle de versão”. Também deve ser automatizado a configuração da aplicação, e os scripts e dados necessários devem ser mantidos no sistema de controle de versão ou no banco de dados.

2. Automatize quase tudo:

Algumas coisas são impossíveis de automatizar, mas, segundo Humble (2014), “ainda não encontraram um processo de compilação ou de implantação que não possa ser automatizado com trabalho e imaginação suficientes”. Muitos processos de compilação podem ser automatizados, até o ponto em que seja necessária a decisão humana, assim como o processo de implantação e para o processo de entrega. Então, devemos automatizar tudo o que puder de forma gradual, com o tempo, conforme os gargalos vão surgindo. No pipeline de implantação a automação é um pré-requisito.

3. Mantenha tudo sobre controle de versão:

Devemos manter em um sistema de controle de versão tudo o que é necessário para compilar, configurar, testar, implantar e entregar uma versão do software. Incluímos também a documentação, scripts de testes, casos de teste automatizados, scripts de configuração, documentação de redes, scripts de implantação e configuração, biblioteca, ferramentas, criação de banco de dados, atualizações e inicializações e documentação técnica, entre outros que a equipe julgar necessário armazenar. Também devemos incluir a versão da aplicação que está instalada em cada um dos ambientes, qual a versão do código e da configuração.

4. Se é difícil, faça com mais frequência e amenize o sofrimento:

Se a entrega da versão é algo sofrido, penoso, faça com mais frequência, sempre que uma mudança ocorrer. Conforme Humble e Farley (2014, p. 27), “se você não pode entregar para os usuários reais toda vez, use um ambiente similar ao de produção depois de cada nova versão”. Se criar a documentação necessária para a aplicação é difícil, faça isso a cada nova mudança ou funcionalidade, para não deixar para o fim. Procure criar uma documentação para cada funcionalidade que satisfaça os princípios de entrega e tente automatizar tudo o que puder.

5. A qualidade deve estar presente desde o início

Princípio baseado no movimento *Lean* “construa com qualidade desde o início”. Portanto, quando mais cedo os defeitos são identificados, mais rápido e barato é para corrigi-los, principalmente se eles não entraram para o controle de versão. Todos na equipe de entrega do software são responsáveis pela qualidade em todos os momentos.

6. Pronto quer dizer versão entregue

O que quer dizer “pronto”? Uma funcionalidade só pode ser considerada pronta quando ela entrega valor ao usuário. Ou seja, pronto significa entrega de uma versão em produção ou quando ela foi demonstrada com sucesso e experimentada por usuários em um ambiente similar ao da produção.

7. Todos são responsáveis pelo processo de entrega

Todos dentro de uma empresa devem estar alinhados com seus objetivos, pois uma equipe pode falhar ou ter sucesso como um time e não como indivíduos. Segundo Humble e Farley (2014, p. 28), “quando algo dá errado, as pessoas perdem muito tempo culpando uma às outras do que corrigindo os defeitos que inevitavelmente surgem em tal abordagem”. Ou seja, encoraja uma colaboração maior entre todos os envolvidos no processo de entrega de software.

8. Melhoria Contínua

A primeira entrega de um software é o primeiro estágio do ciclo de vida. Todos os softwares evoluem e isso significa mais entregas, e é importante que o processo de entrega do software evolua também.



SAIBA MAIS

Controle de versão: a liberdade de excluir

Uma consequência de ter cada versão de cada arquivo sob controle de versão é que isso permite que você seja agressivo em relação a excluir coisas que acha que não irá precisar. Com controle de versão, você pode responder à questão “Devemos excluir esse arquivo?” com “sim!” sem risco; se for a decisão errada, é simples corrigi-la recuperando uma versão anterior. A liberdade de excluir é um grande avanço na manutenção de conjuntos complexos de configuração. Consistência e organização são a chave para que uma equipe grande trabalhe com eficiência. A habilidade de se livrar de ideias e implementações velhas permite que o time experimente coisas novas e melhore o código.

Fonte: Humble (2014, p. 35).

REFLITA



A maioria das pessoas que já se envolveu em um projeto de software próximo da data de entrega de versão sabe o quanto estressante o evento pode ser.
(Humble e Farley)

GERENCIAMENTO DE CONFIGURAÇÃO DE SOFTWARE

Conforme Pressman e Maxim (2016, p. 623), “mudanças são inevitáveis quando o software de computador é construído e podem causar confusão quando os membros de uma equipe de software estão trabalhando em um projeto”. Quando estas mudanças não são bem analisadas antes de serem feitas ou não registradas antes de serem implementadas, surgem as confusões. E para coordenar a confusão, precisamos gerenciar as mudanças.

O Gerenciamento de Configuração de Software (SCM – *Software Configuration Management*) ou GCS é uma atividade de apoio destinada a gerenciar as mudanças, identificando artefatos que precisam ser alterados, as relações entre eles, controle de versão destes artefatos, controlando estas mudanças e auditando e relatando todas as alterações feitas no software. Conforme Humble e Farley (2014, p. 31):

[...] gerência de Configuração é um termo amplamente usado, muitas vezes como sinônimo de controle de versão. Definição formal: Gerência de Configuração se refere ao processo pelo qual todos os artefatos relevantes ao seu projeto, e as relações entre eles, são armazenados, recuperados, modificados e identificados de maneira única. Sua estratégia de gerência de configuração determinará como você gerencia todas as mudanças que ocorrem dentro do seu projeto. Ela registra, assim, a evolução de seus sistemas e aplicações. Ela determinará a forma como seu time colabora – uma consequência vital, mas muitas vezes ignorada, de qualquer estratégia de gerência de configuração.

Para projetos individuais o gerenciamento de configuração de software é útil, pois muitas vezes uma pessoa pode esquecer as mudanças que foram feitas. E para projetos em equipe é essencial, pois vários desenvolvedores trabalham ao mesmo tempo em um sistema, e às vezes, no mesmo local ou as equipes podem estar distribuídas,

com membros em diferentes partes do mundo. Usar um sistema para gerenciar configurações de software garante que as equipes tenham acesso às informações e que uma não interfira no trabalho da outra.

Como as mudanças podem ocorrer a qualquer momento em um software, o gerenciamento de configurações envolve quatro atividades:

Tabela 2 - Atividades de Gerenciamento de Configuração de Software

1. Gerenciamento de Mudanças	Envolve manter o acompanhamento das solicitações dos clientes e desenvolvedores por mudanças no software, definir os custos e o impacto de fazer tais mudanças, bem como decidir se e quando as mudanças devem ser implementadas.
2. Gerenciamento de Versões	Envolve manter o acompanhamento de várias versões de componentes do sistema e assegurar que as mudanças nos componentes, realizadas por diferentes desenvolvedores, não interfiram uma nas outras.
3. Construção do Sistema	É o processo de montagem de componentes de programas, dados e bibliotecas e, em seguida, a compilação e ligação destes, para criar um sistema executável.
4. Gerenciamento de Releases	Envolve a preparação de software para o <i>release</i> externo e manter o acompanhamento das versões de sistema que foram liberadas para uso do cliente.

Fonte: Sommerville (2011, p. 476).

A figura 2 mostra o relacionamento entre as atividades de gerenciamento de configuração de software.

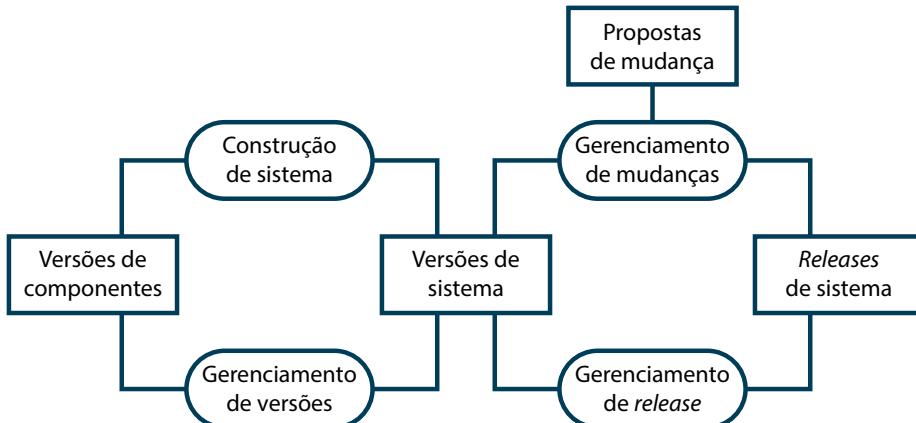


Figura 2 – Atividades de Gerenciamento de Configuração de Software

Fonte: Sommerville (2011, p. 476).

Como o gerenciamento de configurações de software envolve trabalhar com um grande número de informações, é necessário fazer o uso de ferramentas para dar suporte aos processos do SCM. Sobre as ferramentas de gerenciamento de configurações, Sommerville (2011, p. 476) destaca que:

[...] elas vão desde simples ferramentas que oferecem suporte a uma tarefa única de gerenciamento de configuração, tal como acompanhamento de bug, até conjuntos complexos e caros de ferramentas integradas que oferecem suporte a todas as atividades de gerenciamento de configuração. Políticas e processos de gerenciamento de configurações definem como gravar e processar propostas de mudanças de sistema, como decidir quais componentes de sistema alterar, como gerenciar diferentes versões de sistema e seus componentes e como distribuir as mudanças para os clientes. As ferramentas de gerenciamento de configuração são usadas para manter o controle de propostas de alteração, armazenar versões de componentes do sistema, construir sistemas a partir desses componentes e controlar os lançamentos de versões do sistema para os clientes.

No processo temos as *alterações* que podem ocorrer a qualquer momento e por qualquer razão. Com relação às alterações, Pressman e Maxim (2016, p. 624) dizem “qual a origem dessas alterações? A resposta a essa pergunta é variada, assim como as próprias alterações”. A tabela 3 mostra as quatro fontes de alterações fundamentais:

Tabela 3 – Fontes de Alterações fundamentais

FONTES DE ALTERAÇÃO
Novos negócios ou condições de mercado ditam mudanças nos requisitos do produto ou nas regras comerciais
Novas necessidades dos envolvidos demandam modificação dos dados produzidos pelos sistemas de informação, na funcionalidade fornecida ou nos serviços oferecidos.
Reorganização ou crescimento/enxugamento causam alterações em prioridades de projeto ou na estrutura da equipe de engenharia de software.
Restrições orçamentárias ou de cronograma causam a redefinição do sistema ou produto.

Fonte: adaptado de Pressman e Maxim (2016, p. 624).

A SCM é um conjunto de atividades desenvolvidas para controlar as mudanças que podem ocorrer ao longo do ciclo de vida de um software. O Gerenciamento de Configuração de Software pode ser vista como uma atividade de garantia de qualidade do sistema que pode ser aplicado em todo o processo de software (PRESSMAN; MAXIM, 2016).

Um dos problemas do gerenciamento de configurações, segundo Sommerville (2011, p. 476), “é que diferentes empresas falam sobre os mesmos conceitos usando termos diferentes”. A seguir uma tabela com as terminologias usadas no gerenciamento de configuração de software.

Tabela 4 - Terminologias usadas no gerenciamento de configuração de software

TERMO	EXPLICAÇÃO
Item de Configuração ou item de configuração de software (SCI, do inglês <i>Software Configuration Item</i>)	Qualquer coisa associada a um projeto de software (projeto, código, dados de teste, documentos etc.) que tenha sido colocado sob controle de configuração. Muitas vezes, existem diferentes versões de um item de configuração. Itens de configuração têm um nome único.
Controle de Configuração	O processo de garantia de que versões de sistemas e componentes sejam registradas e mantidas para que as mudanças sejam gerenciadas e todas as versões de componentes sejam identificadas e armazenadas por todo o tempo de vida do sistema.
Versão	Uma instância de um item de configuração que difere de alguma forma, de outras instâncias desse item. As versões sempre têm um identificador único, o qual é geralmente composto pelo nome do item de configuração mais um número de versão.
<i>Baseline</i>	Uma <i>baseline</i> é uma coleção de versões de componentes que compõem um sistema. As <i>baselines</i> são controladas, o que significa que as versões dos componentes que constituem o sistema não podem ser alteradas. Isso significa que deveria sempre ser possível recriar uma <i>baseline</i> a partir de seus componentes.

TERMO	EXPLICAÇÃO
<i>Codeline</i>	Uma <i>codeline</i> é um conjunto de versões de um componente de software e outros itens de configuração dos quais esse componente depende.
<i>Mainline</i>	Trata-se de uma sequência de <i>baselines</i> que representam diferentes versões de um sistema.
<i>Release</i>	Uma versão de um sistema que foi liberada para os clientes (ou outros usuários em uma organização) para uso.
Espaço de Trabalho	É uma área de trabalho privada em que o software pode ser modificado sem afetar outros desenvolvedores que possam estar usando ou modificando o software.
<i>Branching</i>	Trata-se da criação de uma nova <i>codeline</i> de uma versão em uma <i>codeline</i> existente. A nova <i>codeline</i> e uma <i>codeline</i> existente podem, então, ser desenvolvidas independentemente.
<i>Merging</i>	Trata-se da criação de uma nova versão de um componente de software, fundindo versões separadas em diferentes <i>codelines</i> . Essas <i>codelines</i> podem ter sido criadas por um <i>branch</i> anterior de uma das <i>codelines</i> envolvidas.
Construção de Sistema	É a criação de uma versão de sistema executável pela compilação e ligação de versões adequadas dos componentes e bibliotecas que compõem o sistema.

Fonte: Sommerville (2011, p. 476).

GERENCIAMENTO DE MUDANÇAS

Para grandes sistemas a mudança é uma realidade, pois os requisitos e as necessidades das empresas mudam durante o ciclo de vida do software, seja por bugs que precisam ser corrigidos ou porque precisam se adaptar às mudanças do mercado. Como controlar essas mudanças no sistema? Com um conjunto de processos de gerenciamento de mudanças e de ferramentas.

Segundo Sommerville (2011, p. 478), “o gerenciamento de mudanças destina-se a garantir que a evolução do sistema seja um processo gerenciado e que seja dada prioridade às mudanças mais urgentes e efetivas”. Está relacionado ao processo de gerenciamento de mudanças a análise de custos e benefícios das mudanças solicitadas pelos clientes, a aprovação dessas mudanças, para ver se valem o investimento e o acompanhamento do que foi alterado no sistema.

Um modelo de processo de gerenciamento de mudanças pode ser conferido na figura 3, que mostra as principais atividades de gerenciamento. Entretanto podem aparecer algumas variantes deste processo, mas para ele ser eficaz, os processos envolvidos sempre devem ter um meio de verificação, custeio e aprovação de mudanças. E este processo entra em vigor quando o sistema for desenvolvido para a entrega aos clientes ou para implantação em uma empresa.

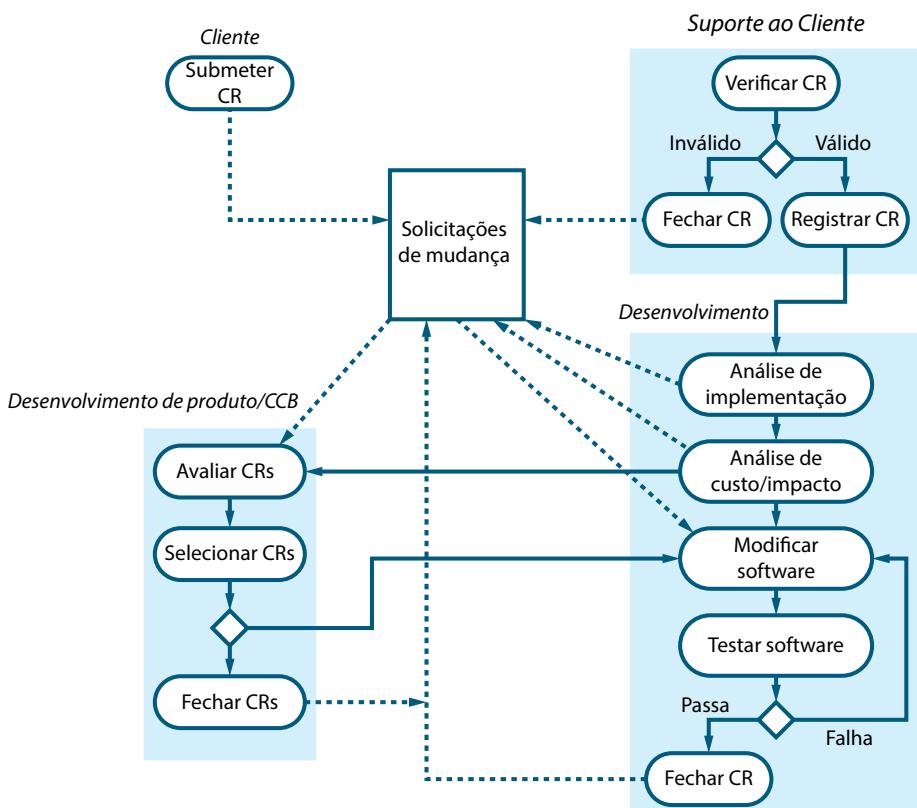


Figura 3 – O processo de gerenciamento de mudanças
Fonte: Sommerville (2011, p. 478).

Quando o cliente preenche e envia uma solicitação de mudança (CR - *Change Request*) com a descrição da mudança requerida, inicia-se o processo de gerenciamento de mudanças. As solicitações de mudanças podem ser novos requisitos ou um relatório de bugs. Algumas empresas podem tratar essas mudanças separadas. As solicitações de mudanças são descritas no formulário de solicitação de alteração (CRF - *Change Request Form*).

Conforme são processadas as solicitações de mudanças, informações são adicionadas ao CR a fim de registrar as decisões tomadas. O CR registra as recomendações feitas a mudança, estimativa de custos, datas, aprovação, implementações e as validações referentes à solicitação de mudanças. Após a solicitação de mudança ter sido submetida, ocorre a análise de sua validade, verificando se todas as solicitações de mudanças foram implementadas. Se for inválido, a solicitação de mudança é fechada e o formulário é atualizado com o motivo de encerramento. Caso a solicitação de mudança for válida, será registrado um CR como solicitação de análise posterior.



REFLITA

Muitas alterações de software são justificadas; portanto, não faz sentido reclamar delas. Em vez disso, esteja certo de ter os mecanismos prontos para cuidar delas.

(Pressman e Maxim)

GERENCIAMENTO DE VERSÕES

O Gerenciamento de Versões (VM - Version Management) segundo Sommerville (2011, p. 480) “é o processo de acompanhamento de diferentes versões de componentes de software ou itens de configuração e os sistemas em que esses componentes são usados”. Também fornece a garantia de que as mudanças feitas por vários desenvolvedores para essas versões não interfiram umas nas outras. O gerenciamento de versões é o processo de gerenciamento de *codelines* e *baselines*.

A figura 4 mostra as diferenças entre *codelines* e *baselines*. Uma *Codelines* é uma sequência de versões do código-fonte do sistema com versões posteriores derivadas das versões anteriores. São aplicadas para componentes de sistema. Uma *Baseline* especifica a versão de cada componente que foi incluído no sistema. Elas são importantes para os casos, em que seja necessário recriar uma versão específica do sistema completo.

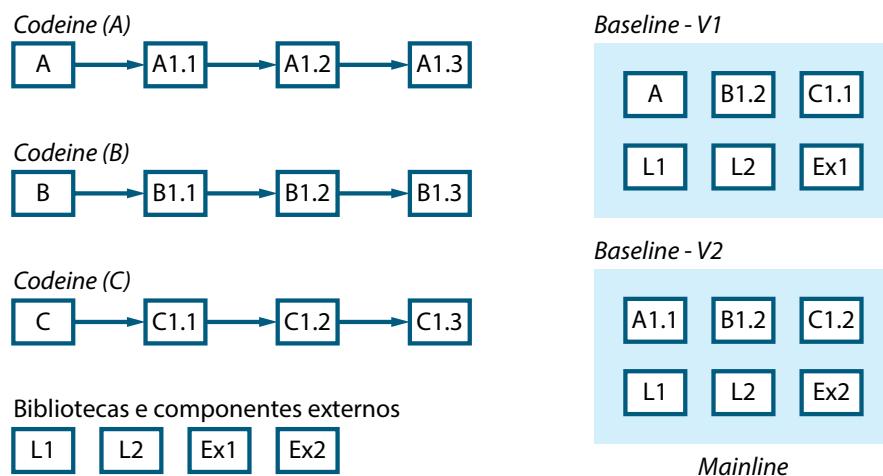


Figura 4 - *Codelines* e *baselines*

Fonte: Sommerville (2011, p. 482).

Na figura 4 podemos observar que diferentes *baselines* usam versões diferentes dos componentes de cada *codeline*. A *Mainline* é uma sequência de versões desenvolvidas a partir de uma *baseline* (original).

Os sistemas de gerenciamento de versões normalmente fornecem alguns recursos, como:

- 1. Identificação de Versão e Release:** as versões gerenciadas recebem identificadores e eles se baseiam no nome do item de configuração seguido por um ou mais números.
- 2. Gerenciamento de Armazenamento:** fornecem recursos de armazenamento para reduzir os espaços de armazenamento requeridos pelas várias versões de componentes.
- 3. Registro de Histórico de Alterações:** todas as mudanças feitas no sistema ou em componentes são registradas e listadas, para que em seguida sejam selecionados os componentes que serão incluídos na *baseline*.

4. **Desenvolvimento independente:** diferentes desenvolvedores podem trabalhar ao mesmo tempo no mesmo componente. O gerenciamento de versões garante que as mudanças feitas nos componentes por desenvolvedores diferentes não interfiram.
5. **Suporte a projetos:** podem ter vários projetos que compartilham componentes e com o suporte a projetos é possível fazer *check-in* e *check-out* (figura 5) dos arquivos dos projetos.

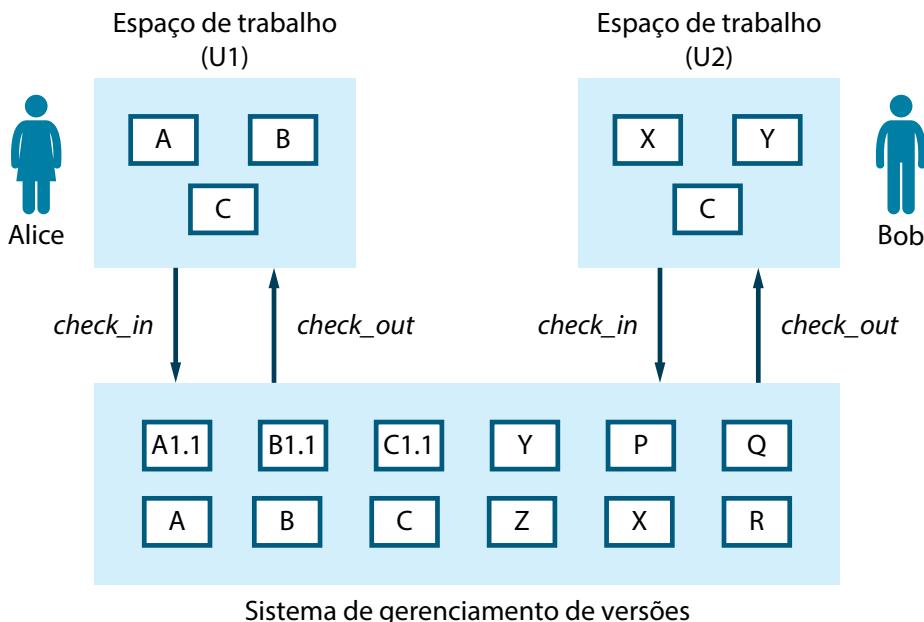


Figura 5 – *Check-in* e *check-out* a partir de um repositório de versões

Fonte: Sommerville (2011, p. 484).

Para o desenvolvimento independente sem interferência, segundo Sommerville usa-se o conceito de repositório público e um espaço de trabalho privado:

[...] os desenvolvedores realizam *check-out* de componentes de um repositório público em seu espaço de trabalho privado e podem mudá-los como quiserem em seu espaço de trabalho privado. Quando as mudanças forem concluídas, eles realizam o *check-in* de componentes para o repositório. Se duas ou mais pessoas estiverem trabalhando em um componente ao mesmo tempo, caso uma deve realizar *check-out* de componente do repositório. Se em um componente for realizado o *check-out* o sistema de gerenciamento de versões normalmente avisará aos outros usuários interessados em realizar o *check-out* desse componente que já foi realizado *check-out* de componente por outra pessoa.

O sistema também garantirá que, quando os componentes modificados são registrados, são atribuídos identificadores de versão diferentes para diferentes versões, as quais são armazenadas separadamente (SOMMERVILLE, 2011, p. 483).

Deste desenvolvimento independente do mesmo componente temos uma consequência que são as codelines que podem se ramificar. Pode haver várias sequências independentes em vez de uma sequência linear de versões que refletem as mudanças para o componente (figura 6). Isso é normal quando se desenvolve sistemas com diferentes desenvolvedores trabalhando independentemente em diferentes versões, fazendo mudanças também de maneiras diferentes.

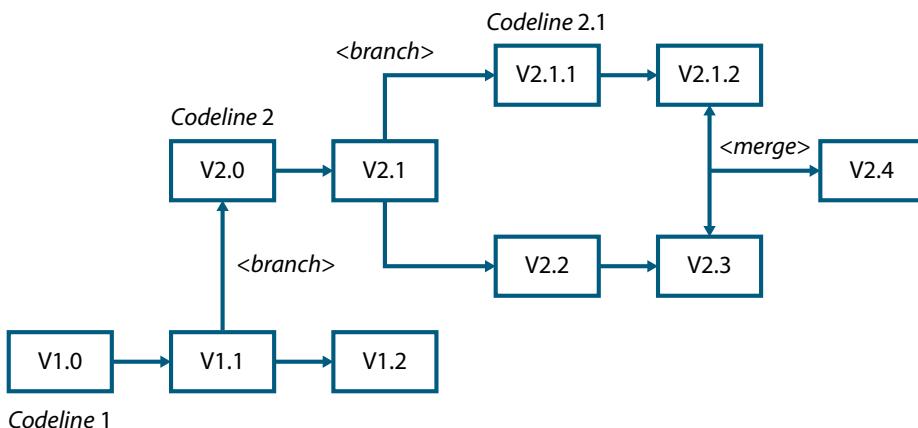


Figura 6 – *Branching e Merging*
Fonte: Sommerville (2011, p. 484).

Pode ser necessário unir as ramificações de codelines em algum momento, a fim de criar uma nova versão de um componente que possui todas as mudanças realizadas. Na figura 6, temos as versões 2.12 e 2.3 do componente que são unidas para criar a versão 2.4. Se as mudanças feitas foram em partes diferentes do código-fonte, as versões dos componentes podem se unir automaticamente. Todavia, frequentemente, podem existir sobreposições entre as mudanças feitas, onde uma interfere na outra. Segundo Sommerville (2011, p. 482) “um desenvolvedor deve verificar se existem conflitos e modificar as mudanças para que estas sejam compatíveis”.

CONSTRUÇÃO DE SISTEMAS

Segundo Sommerville (2011, p. 484), “a construção de sistemas é o processo da criação de um sistema completo, executável por meio da construção e ligação dos componentes de sistemas, bibliotecas externas, arquivos de configuração etc.”. Deve haver uma comunicação entre as ferramentas de gerenciamento de versões e de construção de sistemas, conforme o processo de construção envolve a realização de *check-out* de versões de componentes que se encontram no repositório pelo gerenciamento de versões. Na construção de sistemas também é usada à descrição de configuração que identifica uma *baseline*.

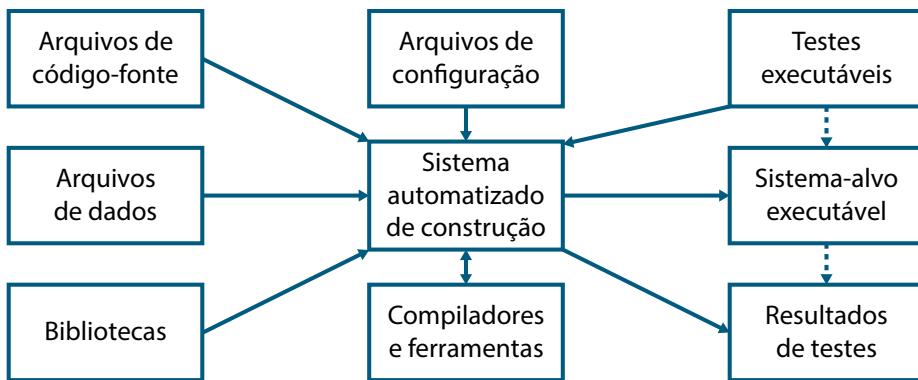


Figura 7 – A Construção de Sistema
Fonte: Sommerville (2011, p. 485).

Segundo Sommerville (2011, p. 485), a construção de sistema envolve:

[...] a montagem de uma grande quantidade de informações sobre o software e seu ambiente operacional. Portanto, para qualquer coisa além dos sistemas muito pequenos, sempre faz sentido usar uma ferramenta de construção automatizada para criar uma construção de sistema. A ideia é você ser capaz de construir um sistema completo com um único comando ou clique.

Na figura 7, podemos notar que não precisamos apenas de arquivos de código-fonte que estão envolvidos na construção do sistema, mas de outros recursos. Podemos ligar os arquivos de código-fonte com as bibliotecas externas, arquivos de dados e a arquivos de configuração que definem a instalação alvo. Pode ser que seja necessário especificar as versões do compilador e outras ferramentas de software que poderão ser usadas.



SAIBA MAIS

É importante fazer uma distinção clara entre suporte de software e gestão de configuração de software. Suporte é um conjunto de atividades de engenharia que ocorrem depois que o software é fornecido ao cliente e posto em operação. Gestão de Configuração é um conjunto de atividades de rastreamento e controle iniciadas quando um projeto de engenharia de software começa e terminadas apenas quando o software sai de operação.

Fonte: Pressman e Maxim (2016, p. 624).

GERENCIAMENTO DE *RELEASES*

Um *release* de um sistema é uma versão de um sistema de software que é distribuída aos clientes. Temos dois tipos de *releases*, os *releases* principais, que fornecem nova e significativa funcionalidade, e o cliente paga por eles. Releases menores que reparam bugs e corrigem os erros relatados pelo cliente. Em geral, eles são distribuídos gratuitamente.

Sobre Sistemas Customizados ou Linhas de Produtos de Software, Sommerville (2013, p. 488) descreve:

[...] o gerenciamento de *releases* de sistema é um processo complexo. Os *releases* especiais do sistema podem precisar ser produzidos para cada cliente e os clientes individuais podem estar executando vários *releases* diferentes do sistema ao mesmo tempo. Isso significa que uma empresa de software vendendo um produto de software especializado pode precisar gerenciar dezenas ou até centenas de diferentes *releases* desse produto. Seus sistemas e processos de gerenciamento de configurações precisam ser projetados para fornecer informações sobre qual *release* do sistema cada cliente tem e o relacionamento entre os *releases* e as versões de sistema. No caso de algum problema, pode ser necessário reproduzir exatamente o software entregue para um determinado cliente.

A preparação e a distribuição de *release* de sistema é considerado um processo caro, e é necessário que a equipe de marketing e publicidade estejam preparadas para convencer os clientes a comprarem o novo *release* do sistema. E isso implica em ter um calendário de *release* bem pensado. Segundo Sommerville

(2011), se os *releases* forem muito frequentes ou requerem muitas atualizações de hardware, os clientes podem não querer mudar, porque terão que pagar por eles. Se os *releases* não forem muito frequentes, os clientes podem querer mudar para um sistema alternativo.

Muitos fatores técnicos e organizacionais devem ser levados em consideração ao pensar em lançar uma nova versão do sistema, conforme a tabela.

Tabela 5 - Fatores que influenciam o planejamento de *release* de sistema

FATOR	DESCRIÇÃO
Qualidade técnica do sistema	Caso sejam relatados defeitos graves de sistema, que afetem a maneira como muitos clientes o usam, pode ser necessário emitir um <i>release</i> de reparação de defeitos. Pequenos defeitos de sistema podem ser reparados mediante a emissão de patches (normalmente distribuídos pela Internet) que podem ser aplicados no <i>release</i> atual do sistema.
Mudanças de plataforma	Talvez você precise criar um novo <i>release</i> de uma aplicação de software quando uma nova versão da plataforma do sistema operacional for lançada.
Quinta Lei de Lehman	Essa "lei" sugere que se você adicionar nova funcionalidade a um sistema, você também introduzirá bugs que limitarão a quantidade de funcionalidade que pode ser incluída no próximo <i>release</i> . Portanto, um <i>release</i> de sistema com funcionalidade nova e significativa pode ser seguido por um <i>release</i> que se concentra em reparar os problemas e melhorar o desempenho.
Concorrência	Para software de mercado de massa, um novo <i>release</i> de sistema pode ser necessário porque um produto concorrente introduziu novos recursos e a fatia de mercado pode ser perdida caso estes não sejam fornecidos aos clientes existentes.
Requisitos de Marketing	O departamento de marketing de uma organização pode ter feito um compromisso para <i>releases</i> estarem disponíveis em uma determinada data.
Propostas de mudanças de cliente	Para sistemas customizados, os clientes podem ter feito e pago por um conjunto específico de propostas de mudanças de sistemas e eles esperam um <i>release</i> de sistema assim que estas sejam implementadas.

Fonte: Sommerville (2011, p. 489).



SAIBA MAIS

O sistema de versões concorrentes (CVS - Concurrent Versions System)

O uso de ferramentas para deter o controle de versão é essencial para uma gestão eficaz das alterações. O sistema de versões concorrentes (CVS) é uma ferramenta largamente utilizada para controle de versão. Projetada originalmente para código-fonte, mas útil para qualquer arquivo baseado em texto, o sistema CVS (1) estabelece um repositório simples, (2) mantém todas as versões de um arquivo sob um único nome de arquivo, armazenando apenas as diferenças entre versões progressivas do arquivo original e (3) protege contra alterações simultâneas de um arquivo, estabelecendo diferentes diretórios para cada desenvolvedor, isolando assim uns dos outros. O CVS mescla as alterações quando cada desenvolvedor completa seu trabalho.

Fonte: Pressman e Maxim (2016, p. 635).

INTEGRAÇÃO CONTÍNUA

Alguns projetos de softwares ficam por longos períodos durante o processo de desenvolvimento em um estado não funcional. Conforme Humble e Farley (2014, p. 55), isso ocorre e muito:

[...] de fato, muito do software desenvolvido por grandes equipes se encontra em um estado basicamente inútil durante a maior parte de seu desenvolvimento. É fácil de entender o motivo disso: ninguém está interessado em executar a aplicação até que esteja terminada. Desenvolvedores introduzem novo código, e podem até rodar os testes automatizados, mas ninguém realmente tenta executar a aplicação em um ambiente parecido com o de produção.

Isso é verdadeiro em projetos que usam *branches* de código de longa duração ou que a equipe adia os testes de aceitação para o final. É comum em projetos, os desenvolvedores programarem as fases de integração para o fim do desenvolvimento, para que consigam ter tempo de unificar os *branches* e colocar o software em estado funcional para que os testes de aceitação possam ser executados. E em outros projetos, ele fica não funcional apenas quando são feitas as mudanças. Qual a diferença entre eles? A diferença entre eles é o uso da Integração Contínua.

Segundo Humble e Farley (2014, p. 55), “integração contínua exige que a aplicação seja compilada novamente a cada mudança feita e que um conjunto abrangente de testes automatizados seja executado”. O objetivo principal da integração contínua é de manter a aplicação em um estado funcional o tempo todo. Para o autor, a integração contínua da aplicação representa uma mudança de paradigma. Sem a integração, a sua aplicação será considerada em estado não funcional, até que ele volte a funcionar normalmente, o que acontece depois dos testes ou da integração. A equipe que utiliza a integração contínua de maneira eficaz entrega o software mais rápido e com menos defeitos, do que as que não usam.

Todavia ao usar a integração contínua, assume-se que a aplicação está em estado funcional (desde que tenha passado por testes automatizados) com cada mudança, pois o desenvolvedor sabe que se parar de funcionar ele pode consertá-lo imediatamente (figura 8).

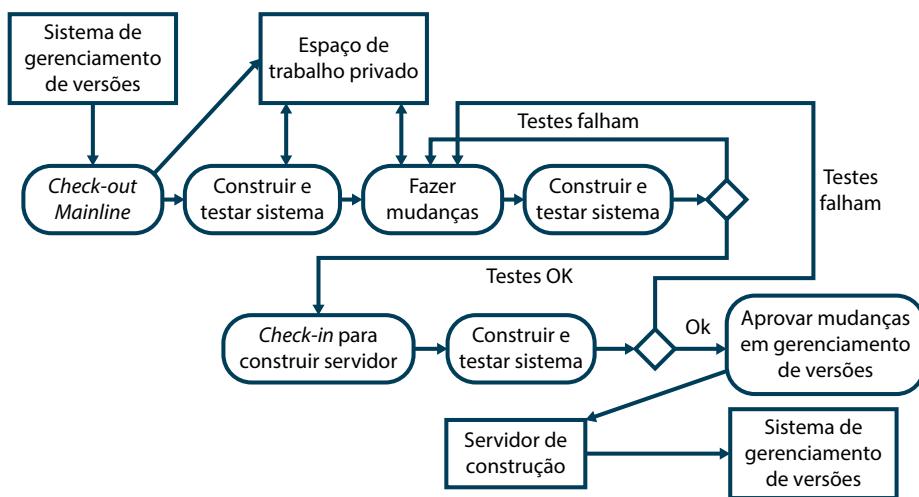


Figura 8 - Integração Contínua

Fonte: Sommerville (2011, p. 486).

De acordo com as Metodologias Ágeis, a integração contínua envolve a frequente reconstrução da *mainline*, após ter sido feita as mudanças no código. A seguir as etapas da integração contínua conforme a figura 8.

Tabela 6 – Etapas da Integração Contínua

1	Realizar <i>check-out</i> do sistema de <i>mainline</i> do sistema de gerenciamento de versões no espaço de trabalho privado do desenvolvedor.
2	Construir o sistema e executar testes automatizados para assegurar que o sistema construído passe em todos os testes. Caso contrário, a construção é interrompida e você deve informar quem realizou o último <i>check-in</i> de sistema de <i>baseline</i> . Eles são responsáveis por reparar o problema.
3	Fazer as mudanças para os componentes de sistema.
4	Construir o sistema no espaço de trabalho privado e executar novamente os testes de sistema. Se os testes falharem, continue a editar.
5	Uma vez que o sistema tenha passado nos testes, verificar isso no sistema construído, mas não aprovar como uma nova <i>baseline</i> de sistema.
6	Construir o sistema no servidor de construção e executar os testes. Você precisará fazer isso para o caso de outras pessoas terem modificado os componentes depois que você tenha feito o <i>check-out</i> do sistema. Se esse for o caso, conferir os componentes que falharam e editá-los, de maneira que os testes passem em seu espaço de trabalho privado.
7	Se o sistema passar em seus testes sobre o sistema de construção, então, aprovar as mudanças feitas como uma nova <i>baseline</i> no <i>mainline</i> de sistema.

Fonte: adaptado de Sommerville (2011, p. 487).



SAIBA MAIS

A primeira pessoa a escrever sobre integração contínua foi Kent Beck, em seu livro *Extreme Programming Explained* (publicado pela primeira vez em 1999). Com todas as outras práticas de Extreme Programming, a ideia da integração contínua é: se a integração frequente do código é boa, por que não fazer isso o tempo todo? No contexto da integração, “o tempo todo” significa a cada vez que alguém introduz uma mudança no código.

Fonte: Humble e Farley (2014, p. 56).

COMO IMPLEMENTAR INTEGRAÇÃO CONTÍNUA

A integração contínua possui alguns pré-requisitos para a sua prática. Mas antes de abordar estes pré-requisitos, vamos conhecer o que precisamos ter antes de implementar a integração contínua.

1. **Controle de Versão:** precisamos ter um único repositório para guardar tudo do projeto, como: códigos, testes, scripts de banco de dados, de compilação e implantação, cenários, instruções de instalação etc.
2. **Um processo Automatizado de Compilação:** devemos executar o processo a partir da linha de comando, como compilar a aplicação e rodar testes ou rodar um sistema de múltiplos estágios que chamam outros scripts que executam outras tarefas. É necessário que se execute o processo de compilação automaticamente em seu ambiente de integração contínua.
3. **Aceitação da Equipe:** segundo Humble e Farley (2014, p. 57), “integração contínua é uma prática, não uma ferramenta. Ela requer um alto grau de comprometimento e disciplina da equipe de desenvolvimento”. Assim, se torna necessário que todos da equipe aceitem e se acostumem a fazer *check-in* de novas funcionalidades. Se a equipe não adotar esta prática, qualquer tentativa de integração contínua, poderá não levar a melhorias de qualidade desejada.

Pré-requisitos para a Integração contínua:

A seguir, as práticas que devem ser seguidas para a prática da integração contínua.

- **Check-ins regulares:** fazer *check-ins* regulares para o trunk de desenvolvimento é uma das práticas mais importantes da integração contínua. Eles devem acontecer ao longo do dia, pois torna suas mudanças menores e reduz a probabilidades de falharem. Caso ocorram mudanças incorretas, terá uma versão anterior do software se precisar reverter essas mudanças.
- **Crie um conjunto de testes automatizados abrangentes:** o ideal é a criação de um conjunto abrangente de testes automatizados para o processo de integração contínua: testes unitários, testes de componentes e testes de aceitação.
- **Mantenha o processo de compilação e de testes curto:** tempo de compilação do código e dos testes unitários não deve levar muito tempo, para não ter problemas, como: parar de rodar a compilação por completo e de executar os testes somente quando fazem *check-in*. A integração contínua demorará muito que não se saberá qual *check-in* quebrou a compilação, se esse for o caso. Com essa demora, as pessoas não farão *check-in* com frequência.

- **Como gerenciar seu espaço de trabalho de desenvolvimento:** é importante para os desenvolvedores que seu espaço de trabalho seja cuidadosamente gerido, tanto para a produtividade quanto para a sanidade.

PRÁTICAS ESSENCIAIS

Agora vamos conhecer algumas práticas consideradas essenciais para o funcionamento da integração contínua.

- **Não faça *check-ins* se o processo de compilação estiver quebrado:** não introduza código novo quando o processo de compilação estiver quebrado. Se a compilação estiver quebrada, o desenvolvedor deve estar preparado para consertá-la.
- **Sempre rode os testes de commit localmente antes de *check-in*, ou use o servidor de IC para isso:** rode os testes localmente antes de executar alguma coisa, é uma garantia que realmente vai funcionar. Um *check-in* cria uma versão candidata e com isso, fica fácil testar antes de publicar a versão definitiva.
- **Espere que os testes obtenham sucesso antes de continuar:** a integração contínua é um recurso compartilhado por todos na equipe. Quando a equipe usa a integração contínua, qualquer quebra na compilação é um inconveniente para todos na equipe e para o projeto todo. Somente após o *check-in* ter sido compilado e passado pelos testes, os desenvolvedores podem começar alguma tarefa nova.
- **Nunca vá para casa com um processo de compilação quebrado:** pois poderá não lembrar com clareza as mudanças que foram feitas quando voltar, demorando ainda mais para corrigir o problema. Por isso, o *check-in* deve ser feito regularmente e cedo, para que se tenha tempo de corrigi-lo, em casos de quebra na compilação. Não se deve realizar *check-ins* uma hora antes do fim do expediente dos desenvolvedores.
- **Esteja sempre preparado para voltar à revisão anterior:** todos cometemos erros, apesar dos esforços ou às vezes o processo de compilação pode quebrar. Nessas circunstâncias, caso não consiga consertar o problema rapidamente, deve-se voltar à versão anterior e fazer as devidas correções.
- **Limite o tempo antes de reverter:** estabeleça uma regra para controlar o tempo de conserto dos erros. Caso o tempo seja extrapolado, reverta para a versão anterior.

- **Não comente testes que estão falhando:** quando testes bem-sucedidos começam a falhar, pode ser difícil entender o porquê e descobrir o que pode estar acontecendo, pode requer um esforço considerável. Então, não comente testes que estão falhando para que o *check-in* possa ser feito, somente como último recurso.
- **Assuma a responsabilidade pelas quebras causadas por suas mudanças:** se você introduziu uma mudança no código, pode ser que essa mudança faça com que a compilação quebre, é sua responsabilidade corrigir.
- **Desenvolvimento guiado por testes:** é necessário que seja feito um conjunto abrangente de testes que são essenciais para a integração contínua. E uma maneira eficaz de obter uma cobertura é por meio de TDD (*Test-Driven Development*) ou desenvolvimento guiado por testes. O TDD é uma abordagem iterativa usada para desenvolver software e seu princípio básico se baseia em escrever testes automatizados para a funcionalidade antes de ser implementada.



SAIBA MAIS

Testes unitários são escritos para testar o comportamento de pequenas partes de sua aplicação em isolamento. Eles podem ser executados sem a necessidade de executar a aplicação por completo. Testes de componentes testam o comportamento de vários componentes da aplicação. Como os testes unitários, geralmente, não exigem que a aplicação seja executada por completo. Testes de aceitação garantem que a aplicação satisfaz os critérios de aceitação decididos pelo negócio, incluindo tanto a funcionalidade que a aplicação deve oferecer como outras características, como capacidade, disponibilidade, segurança e assim por diante.

Fonte: Humble e Farley (2014, p. 60).

Para finalizar este tópico, podemos dizer que a ideia principal da integração contínua é a diminuição dos problemas e riscos por meio de um melhor monitoramento das mudanças e da integração frequente do código. Portanto, caso ocorram falhas, a integração contínua ajuda para que o impacto seja menos significativo e facilita a identificação e correção dessas falhas.



CONSIDERAÇÕES FINAIS

Prezado(a) aluno(a), nesta unidade vimos os princípios da Entrega Contínua de Software, os problemas da entrega que podem ocorrer e quais as práticas necessárias para realizá-la na empresa de forma rápida, com poucos impactos e com qualidade.

Também discutimos o problema de entregar software e como podemos otimizar essa entrega sem impactar no orçamento e na qualidade, ainda que com prazos cada vez menores, já que muitas novas funcionalidades são adicionadas ou as que já existem são melhoradas e os defeitos encontrados são corrigidos.

Falamos sobre a Entrega Contínua (*Continuous Delivery*) que é o processo de implantação contínua em ambiente de produção e aprendemos que o seu objetivo principal é encontrar formas de entregar software com valor para o cliente com qualidade de forma eficiente, rápida e confiável.

Aprendemos sobre os pré-requisitos necessários para entendermos o processo de *pipeline* de implantação e de alguns antipadrões que muitas equipes de desenvolvimento de software tem, além disso, descobrimos que a entrega de software possui alguns princípios que devem ser seguidos para que o processo de entrega da versão seja eficaz.

Caminhando para o final de nossa unidade, falamos sobre o Gerenciamento de Configuração de Software (SCM – *Software Configuration Management*) ou GCS, seus conceitos e princípios. O GSC é uma atividade de apoio para gerenciar as mudanças, identificando artefatos que precisam ser alterados, as relações entre eles, controle de versão destes artefatos, controlando estas mudanças e auditando e relatando todas as alterações feitas no software.

E, finalizando, entendemos alguns conceitos básicos sobre a Integração contínua e seus princípios, pois a ideia principal da integração contínua é a diminuição dos problemas e riscos por meio de um melhor monitoramento das mudanças e da integração frequente do código.

Depois desta Unidade, com o conhecimento que já adquirimos sobre Entrega Contínua de *software* e seus conceitos e princípios, podemos passar para a próxima Unidade para que assim você se aprofunde ainda mais no conhecimento. Preparados? Então vamos em frente! Bons estudos!

ATIVIDADES



1. "A entrega contínua existe para que as funcionalidades sejam liberadas continuamente e de forma segura para o cliente. Imagine que ao submeter um arquivo para o repositório de código-fonte, o ambiente de homologação seja automaticamente atualizado e o ajuste seja disponibilizado para testes sem a necessidade de intervenção manual". (DEVMEDIA, [2018], on-line)¹.

Com base no que foi exposto, explique o padrão *Pipeline de implantação*?

2. A entrega do sistema geralmente é feita pelo grupo de desenvolvedores. E frequentemente o processo de entrega é dominado pela equipe de desenvolvimento. Pensando sobre isso, descreva os três objetivos do *pipeline* de implantação.

3. A Entrega Contínua ajuda que o software responda de forma rápida às expectativas dos clientes, aumentando a qualidade dos seus produtos a um baixo custo. A entrega de software possui princípios que devem ser seguidos para que o processo de entrega da versão seja eficaz. Com base nesta informação, assinale as alternativas que mostre os princípios de entrega de software:

- I. Criar um processo de confiabilidade e repetitividade de entrega de versão e Automatizar quase tudo.
- II. Mantenha tudo sobre controle de versão, e se for difícil, faça com menos frequência e amenize o sofrimento.
- III. A qualidade deve estar presente desde o início e pronto quer dizer versão entregue
- IV. Todos são responsáveis pelo processo de entrega.

É correto o que se afirma em:

- a) I.
- b) I e IV.
- c) I, II e III.
- d) I, III e IV.
- e) I, II, III e IV.

ATIVIDADES



4. Para Pressman (2016), as mudanças são inevitáveis quando o software de computador é construído e podem causar confusão quando os membros de uma equipe de software estão trabalhando em um projeto. Pensando sobre esse assunto, complete as lacunas:

_____ é um termo amplamente usado, muitas vezes como sinônimo de _____. Sua estratégia de _____ determinará como você gerencia todas as mudanças que ocorrem dentro do seu projeto. Ela registra, assim, a evolução de seus sistemas e aplicações.

- a) Gerência de Configuração, Gerência de Projetos, gerência de configuração.
 - b) Gerência de Configuração, Gerência de Processos, Controle de versão de configuração.
 - c) Gerência de Configuração, Gerência de Configuração, gerência de configuração.
 - d) Gerência de Configuração, Controle de Versão, gerência de configuração.
 - e) Gerência de Configuração, Gerência de Projetos, Controle de Versão.
5. É Importante fazer uma distinção clara entre suporte de software e gestão de configuração de software. Suporte é um conjunto de atividades de engenharia que ocorrem depois que o software é fornecido ao cliente e posto em operação. Pensando sobre isso, descreva o gerenciamento de *releases* de sistemas sobre os Sistemas Customizados ou Linhas de Produtos de Software.



ENTREGA CONTÍNUA: UM ESTUDO DE CASO PARA AUTOMATIZAÇÃO DO FLUXO DE IMPLANTAÇÃO DO SISTEMA INTEGRA

A realização de entregas manuais de *software* pode expor a organização e o usuário final a riscos e gerar um grande desconforto para quem está implantando um sistema em produção. Em muitos ambientes de desenvolvimento os processos realizados, entre a solicitação de uma manutenção e a implantação da mesma em produção, faz com que a duração de um ciclo atinja semanas, meses ou até mesmo mais de um ano. Quanto maior sua duração, maior será o risco de ocorrerem erros durante a implantação.

A automatização dos processos diminui o tempo entre um código pronto e seu uso pelos usuários finais. Assim, o risco associado à entrega tem uma redução significativa, permitindo a reversão de mudanças com facilidade e rápida obtenção de *feedback* [...]. Essa automatização pode ser obtida através da integração de todas as atividades da entrega de *software*. Entre elas está o Controle de Modificações, o Controle de Versões, o Controle de Gerenciamento de Construção, os Testes Automatizados, a Análise Estática do Código, a Gerência de Dependências, a Gerência de Artefatos e a Gerência de Infraestrutura. Para que essas atividades sejam transformadas em uma etapa normal e contínua é fundamental que todos os membros da equipe trabalhem em conjunto, isso inclui desde desenvolvedores e testadores, até times de implantação e operação [...].

A Entrega Contínua (*Continuous Delivery*) surgiu para reduzir o custo, o tempo e o risco da entrega de *software*. Para isso os processos devem ser amparados por ferramentas e *scripts* de automatização que garantam que o *software* estará pronto para ser implantado em produção a qualquer momento [...]. Para apoiar o processo de Entrega Contínua, a Gerência de Configuração de *Software* (GCS) fornece técnicas, processos e ferramentas para automatizar e implantar *software* em ciclos curtos.

A configuração de *software* é composta por um conjunto de itens de configuração. Entre eles estão documentos, programas, dados e ambiente de desenvolvimento. Todos esses itens devem ser controlados pela GCS, que aplicada ao longo de todo o ciclo de vida da aplicação. Seu objetivo é identificar, controlar, auditar e relatar modificações que ocorrem durante o desenvolvimento e manutenção do *software*.

A Integração Contínua é uma atividade da GCS que consiste em diminuir o ciclo de integração, tornando-o o mais frequente possível. Além disso, um *feedback* contínuo pode ser obtido através de práticas e ferramentas que auxiliam na verificação e correção de erros de integração assim que eles são introduzidos, quando ainda é barato corrigi-los [...]. O *pipeline* de implantação, que será apresentado mais a frente, é utilizado para levar o princípio de IC à sua conclusão lógica. Ao combiná-lo com o uso correto da GCS e um alto grau de testes automatizados é possível realizar entregas em ambiente de teste, desenvolvimento ou produção, a partir de um clique em um botão.



O *pipeline* de implantação é uma implementação de ponta a ponta da automatização do processo de compilação, testes e implantação, que permite às equipes de teste e operação implantar a aplicação em ambientes de teste, de homologação e de produção com o apertar de um botão. Aos desenvolvedores, é permitida a visualização dos estágios do processo de entrega alcançados por cada versão do *software* e dos problemas encontrados em cada uma delas. Além disso, proporciona ao gerente a verificação e monitoração de métricas fundamentais como tempo de ciclo, taxa de transferência e qualidade de acesso, permitindo a visibilidade sobre o processo de entrega que torna possível a identificação, otimização e remoção de gargalos.

Testar é uma atividade multifuncional que deve ser executada continuamente desde o começo do projeto, por todos os integrantes do time. Para garantir a qualidade, a cada mudança realizada na aplicação, em sua configuração ou no ambiente de *software*, devem ser escritos testes em múltiplos níveis (unitários, de componentes e de aceitação) e incluídos no *pipeline* de implantação para que sejam executados. A execução sistemática de testes tem o objetivo específico de encontrar e remover o maior número possível de erros, evitando assim que eles sejam encontrados pelo cliente.

O principal objetivo do *pipeline* de implantação é permitir que todos os envolvidos no processo de entrega tenham visibilidade sobre o progresso dos processos, dessa forma, torna-se possível identificar quais mudanças quebraram a aplicação e quais se tornaram versões candidatas apropriadas para a execução de testes manuais e entrega. Quando uma versão passa por todo o *pipeline* com sucesso, significa que ela foi submetida a testes automatizados e manuais em ambiente similares aos de produção.

Fonte: Almeida e Silva (2015).

MATERIAL COMPLEMENTAR



LIVRO

Entrega Contínua - Como Entregar Software de Forma Rápida e Confiável

Jez Humble, David Farley

Editora: Bookman

Sinopse: entregar uma nova versão de software para usuários costuma ser um processo cansativo, arriscado e demorado. Entretanto por meio da automação dos processos de compilação, implantação e teste, e da colaboração otimizada entre desenvolvedores, testadores e a equipe de operações, times de entrega podem lançar mudanças em questão de horas – algumas vezes, em minutos. Neste livro, Jez Humble e David Farley apresentam os princípios, as práticas e as técnicas de ponta que tornam possível uma entrega rápida e de alta qualidade, independentemente do tamanho do projeto ou da complexidade de seu código.



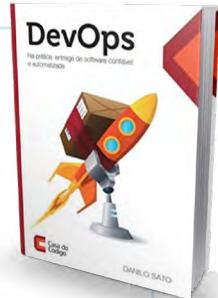
LIVRO

DevOps na Prática: Entrega de Software Confiável e Automatizada

Daniilo Sato

Editora: Casa do Código

Sinopse: entregar software em produção é um processo que tem se tornado cada vez mais difícil no departamento de TI de diversas empresas. Ciclos longos de teste e divisões entre as equipes de desenvolvimento e de operações são alguns dos fatores que contribuem para este problema. Mesmo equipes ágeis que produzem software entregável ao final de cada iteração sofrem para chegar em produção quando encontram estas barreiras. DevOps é um movimento cultural e profissional que está tentando quebrar essas barreiras. Com o foco em automação, colaboração, compartilhamento de ferramentas e de conhecimento, DevOps está mostrando que desenvolvedores e engenheiros de sistema têm muito o que aprender uns com os outros.



MATERIAL COMPLEMENTAR



NA WEB

O que é DevOps afinal?

Artigo que fala sobre a história do DevOps, sua origem, conceitos, ferramentas, ambiente e a infraestrutura. DevOps ainda é um movimento em constante construção e definição, eu citei uma série de melhorias que fazem parte da cultura DevOps, cabe a cada empresa ou a cada gestor estudar e descobrir a melhor forma de combinar essas pequenas receitas técnicas e aplicar em seu ambiente.

Para saber mais, acesse o link: <<http://gutocarvalho.net/octopress/2013/03/16/o-que-e-um-devops-afinal/>>.



NA WEB

Integração Contínua para Testadores

Post que fala sobre o que é Integração Contínua e como ela é utilizada dentro de um ambiente de desenvolvimento e como ela pode ser utilizada por nós, testadores.

Para ficar por dentro do assunto acesse o link: <<http://www.qualister.com.br/blog/integracao-continua-para-testadores---parte-1>>.



REFERÊNCIAS

ALMEIDA, A. L. S. C.; SILVA, E. O. **Entrega Contínua:** Um Estudo de Caso para Automatização do Fluxo de Implantação do Sistema Integra. Juiz de Fora: Centro de Ensino Superior de Juiz de Fora, 2015.

HUMBLE, J.; FARLEY, D. **Entrega Contínua:** Como Entregar Software De Forma Rápida e Confiável. Porto Alegre: Bookman, 2014.

PRESSMAN, R.; MAXIM, B. R. **Engenharia de Software – Uma abordagem profissional.** 8. Ed. Porto Alegre: AMGH, 2016.

SOMMERVILLE, I. **Engenharia de Software.** 9.ed. - São Paulo: Pearson Prentice Hall, 2011.

REFERÊNCIA ON-LINE

¹Em: <<https://www.devmedia.com.br/entrega-continua-de-software-revista-net-magazine-100/26312>>. Acesso em: 7 maio 2018.



GABARITO

1. *Pipeline de implantação*: em essência, uma implementação automatizada do processo de compilar todas as partes de uma aplicação, implantá-la em um ambiente qualquer – seja de homologação ou produção – testá-la e efetuar sua entrega final. Cada organização tem implementações diferentes de seu pipeline de implantação, dependendo de sua cadeia de valor para entregar software, mas os princípios que regem o pipeline são os mesmos.
2. Os três objetivos do pipeline de implantação são:
 - Tornar cada parte do processo de compilação, implantação, teste e entrega de versão visível a todos os envolvidos promovendo a colaboração das equipes.
 - Melhorar o feedback do processo, identificando e resolvendo os problemas o mais cedo possível.
 - Permitir que as equipes entreguem e implantem qualquer versão do software em qualquer ambiente, a qualquer momento por meio de um processo completamente automatizado.
3. Alternativa D.
4. Alternativa C.
5. O gerenciamento de *releases* de sistema é um processo complexo. Os *releases* especiais do sistema podem precisar ser produzidos para cada cliente e os clientes individuais podem estar executando vários *releases* diferentes do sistema ao mesmo tempo. Isso significa que uma empresa de software vendendo um produto de software especializado pode precisar gerenciar dezenas ou até centenas de diferentes *releases* deste produto. Seus sistemas e processos de gerenciamento de configurações precisam ser projetados para fornecer informações sobre qual *release* do sistema cada cliente tem e o relacionamento entre os *releases* e as versões de sistema.



DESENVOLVIMENTO DE SISTEMAS

Objetivos de Aprendizagem

- Entender o que são Sistemas Críticos.
- Compreender o que é a Engenharia de Segurança.
- Conhecer a respeito de Reengenharia e a Manutenção de Software.
- Estudar os conceitos básicos da Engenharia Reversa.

Plano de Estudo

A seguir, apresentam-se os tópicos que você estudará nesta unidade:

- Sistemas Críticos
- Engenharia de Segurança
- Reengenharia e Manutenção de Software
- Engenharia Reversa

INTRODUÇÃO

Olá, aluno(a)! Esta unidade tem por objetivo introduzir conceitos sobre os Sistemas Críticos, Engenharia de Segurança, Reengenharia e Manutenção e Engenharia Reversa.

Vamos começar falando sobre o que é um Sistema Crítico e seus tipos. A confiança é a propriedade mais importante em um sistema crítico. Abordaremos sobre os conceitos de um Sistema Sociotécnico – que inclui pessoas, software e hardware – mostrando o que é necessário para ter uma perspectiva de proteção e uma confiança no software. Falaremos sobre falhas, confiança e a proteção de software e como elas não devem ser tratadas isoladamente e como elas podem afetar drasticamente um sistema.

Outro tópico abordado nesta unidade é sobre a Engenharia de Segurança. A segurança é considerada um dos aspectos importantes da garantia da qualidade do software. Engenheiros de software devem conhecer essas ameaças para que consigam ter a capacidade de proteger os sistemas desenvolvidos.

Ainda vamos estudar sobre a Reengenharia e a Manutenção de Software. A evolução do sistema envolve a compreensão e o conhecimento do programa que tem que ser mudado, para a implementação de novas mudanças. A reengenharia pode melhorar a manutenibilidade do sistema, mas o sistema reconstruído provavelmente não será de fácil manutenção como um novo sistema. O software continuará evoluindo com o passar do tempo, independentemente do domínio de aplicação, tamanho ou da sua complexidade.

Vamos também estudar a Engenharia Reversa. Ela é um processo de recuperação do projeto, onde as ferramentas extraem informações do projeto de dados, da arquitetura e procedural com base em um programa já desenvolvido. As informações podem ser extraídas do projeto do código-fonte pela engenharia reversa.

Preparado(a) para começar a leitura? Então, vamos seguir em frente. Boa leitura e bons estudos.

A close-up photograph of a silver computer keyboard. A single key in the top right corner is highlighted in red and features the word "CRITICAL" in white capital letters. The surrounding keys are standard white with black or blue text.

SISTEMAS CRÍTICOS

Hoje, as falhas de softwares podem ocorrer a qualquer momento e são, segundo Sommerville (2007, pg. 29), relativamente comuns, pois “na maioria dos casos, essas falhas causam inconveniências, mas não danos sérios no longo prazo”. Entretanto, em alguns sistemas, certas falhas podem resultar em danos mais sérios e significativos, como perdas econômicas ou danos físicos. Esse tipo de sistema dos quais as pessoas ou os negócios dependem são chamados de **Sistemas Críticos**. São sistemas técnicos ou sociotécnicos que se falharem ao desempenhar as suas funções, podem causar sérios problemas e prejuízos significativos.

Temos três tipos principais de Sistemas Críticos:

- 1. Sistemas Críticos de Segurança:** nesse tipo de sistema a falha pode resultar em prejuízos, danos sérios ao meio ambiente e perda de vida humana. Exemplo: Sistema de Controle de uma Fábrica de Produtos Químicos, Sistemas Médicos.
- 2. Sistemas Críticos de Missão:** nesse tipo de sistema a falha resulta em problemas de alguma atividade que possui metas. Exemplo: Sistema de Navegação de uma nave espacial.
- 3. Sistemas Críticos de Negócios:** nesse tipo de sistema a falha resulta em custos altos as empresas que usam esse sistema. Exemplo: Sistema de Contabilidade, Sistema Bancário.

Os sistemas de software não são sistemas isolados, mas componentes essenciais de sistemas mais abrangentes com algum propósito humano, social ou organizacional. Por exemplo, o software de sistema de controle meteorológico no deserto controla os instrumentos em uma estação meteorológica. Ele se comunica com outros sistemas de software e é uma parte de sistemas de previsão meteorológica nacional e internacional mais amplos. Além do hardware e do software, esses

sistemas incluem processos para a previsão do tempo e para as pessoas que operam o sistema e analisam seus resultados. Também incluem as organizações que dependem do sistema para ajudar na previsão do tempo para os indivíduos, governos, indústria etc. Esses sistemas mais amplos são chamados sistemas sociotécnicos. Eles incluem elementos não técnicos, como pessoas, processos, regulamentos etc., bem como componentes técnicos, computadores, software e outros equipamentos.

Confiança é a propriedade mais importante de um sistema crítico. Confiança é um termo que abrange os atributos: disponibilidade, confiabilidade, segurança e proteção. Para que a confiança seja a propriedade mais importante de um sistema crítico temos algumas razões:

- Os sistemas que não são confiáveis, seguros ou sem proteção são normalmente rejeitados pelos usuários.
- Os custos de falha nesses sistemas são muito altos.
- Os sistemas não confiáveis podem causar perda de informações ou danos físicos aos seus usuários.

Para Sommerville (2007, pg. 30), “o alto custo de falha de sistemas críticos significa que métodos e técnicas confiáveis devem ser usados para seu desenvolvimento”. E consequentemente, são desenvolvidos usando técnicas consagradas, ou seja, mais tradicionais em vez de técnicas mais recentes. Os desenvolvedores de sistemas críticos são naturalmente mais conservadores, e, portanto, preferem as técnicas mais antigas, pois conhecem seus pontos fortes e fracos. As técnicas mais recentes, aparentemente melhores, mas não conhecem os problemas que podem surgir a longo prazo.

Poucos sistemas críticos podem ser totalmente automatizados, a grande maioria é constituída de sistemas sociotécnicos, nos quais as pessoas controlam e monitoram as atividades e operações dos sistemas. O custo de uma falha em um sistema crítico é alto, e as pessoas que realizam as atividades nesses sistemas precisam enfrentar situações não esperadas e contornar dificuldades. E por isso, o custo de verificação e validação de sistemas críticos é altíssimo, e representam em torno de 50% dos custos totais do desenvolvimento do sistema.

Segundo Sommerville (2011, pg. 185), “sistemas sociotécnicos são tão complexos que é praticamente impossível entendê-los como um todo”. Para entender, o melhor é percebê-los como camadas (figura 1).



Figura 1 - A pilha de sistemas sociotécnicos

Fonte: Sommerville (2011, p. 185).

As camadas fazem parte da pilha de sistemas sociotécnicos, conforme a tabela a seguir:

Tabela 1 – As camadas da Pilha de Sistemas Sociotécnicos

CAMADAS DA PILHA DE SISTEMAS SOCIOTÉCNICOS	
1. A camada de equipamentos	É composta de dispositivos de hardware, alguns dos quais podem ser computadores.
2. A camada de sistema operacional	Interage com o hardware e fornece um conjunto de recursos comuns para as camadas superiores de software no sistema.
3. A camada de comunicações e gerenciamento de dados	Estende-se até os recursos do sistema operacional e fornece uma interface que permite interação com a mais ampla funcionalidade, como o acesso a sistemas remotos, o acesso ao banco de dados de sistema etc. Algumas vezes, ela é chamada <i>middleware</i> , já que está entre a aplicação e o sistema operacional.
4. A camada de aplicação	Fornecer a funcionalidade específica da aplicação que é requerida. Nela, podem haver muitos programas diferentes de aplicação.
5. A camada de processos de negócio	Nesse nível são definidos e aprovados os processos do negócio da organização que usam o sistema de software.

CAMADAS DA PILHA DE SISTEMAS SOCIOCÉNICOS	
6. A camada organizacional	Essa camada inclui processos de alto nível estratégico, bem como regras de negócio, políticas e normas que devem ser seguidas ao se usar o sistema.
7. A camada social	Nessa camada estão definidas as leis e os regulamentos da sociedade que governa o funcionamento do sistema.

Fonte: adaptado de Sommerville (2011, p. 185).

A interação ocorre entre as camadas vizinhas, onde cada camada superior oculta os detalhes da camada inferior. Entretanto, conforme Sommerville (2011), na prática nem sempre é assim, pois pode haver interações inesperadas entre as camadas, o que pode ocasionar problemas para o sistema todo, como o exemplo citado pelo autor:

[...] por exemplo, digamos que haja uma mudança na lei que regula o acesso às informações pessoais. Essa alteração vem da camada social e leva à necessidade de novos procedimentos organizacionais e de mudanças nos processos de negócios. No, entanto, o sistema de aplicação pode não ser capaz de fornecer o nível de privacidade exigido para que as alterações sejam implementadas na camada de comunicação e gerenciamento de dados (SOMMERVILLE, 2011, 185).



SAIBA MAIS

O termo ‘sistema’ é usado universalmente. Falamos de sistemas de computador, sistemas operacionais, sistemas de pagamento, sistema de ensino, sistema de governo e assim por diante. Essas são aplicações, obviamente muito diferentes da palavra ‘sistema’, apesar de compartilharem a característica de ser, de alguma forma, mais do que simplesmente a soma de suas partes. Entretanto o foco é na discussão em sistemas que incluem computadores e que tenham algum objetivo específico, como permitir a comunicação, apoiar a navegação ou calcular salários. Uma definição útil desses tipos de sistemas é a seguinte: um sistema é uma coleção intencional de componentes inter-relacionados, de diferentes tipos, que funcionam em conjunto para atingir um objetivo. Essa definição geral abrange uma vasta gama de sistemas. Por exemplo, um sistema simples, como o apontador a laser, pode incluir alguns componentes de hardware e pouco além de uma pequena quantidade de software de controle. Em contraste, um sistema de controle aéreo inclui milhares de componentes de hardware e software, além de usuários humanos que tomam decisões com base em informações do sistema computacional.

Fonte: Sommerville (2011, p. 186).

SISTEMAS SOCIOTÉCNICOS

Os Sistemas sociotécnicos, segundo Sommerville (2011, pg. 187), “são sistemas corporativos destinados a contribuir para o cumprimento de uma meta de negócios”. Esta meta pode ser um aumento nas vendas em caso de um sistema de vendas, redução de materiais usados na indústria de manufatura, cálculos de arrecadação de impostos, manter um espaço aéreo seguro etc. Ainda sobre os sistemas sociotécnicos, Sommerville descreve que:

[...] como os sistemas sociotécnicos estão embutidos em um ambiente organizacional, a aquisição, o desenvolvimento e o uso desses sistemas são influenciados pelas políticas de procedimentos organizacionais, bem como por sua cultura de trabalho. Os usuários do sistema são pessoas influenciadas pela forma como a organização é gerida e suas interações com outras pessoas dentro e fora da organização. Quando você está tentando desenvolver sistemas sociotécnicos, precisa entender o ambiente organizacional em que eles serão usados. Se você não fizer isso, os sistemas podem não atender às necessidades de negócios, e os usuários e seus gerentes podem rejeitar o sistema (SOMMERVILLE, 2011, 187).

Alguns fatores organizacionais do ambiente do sistema podem afetar os requisitos, o projeto e operação de um sistema sociotécnico, como:

- **Mudanças de processos:** o sistema exige mudanças nos processos de trabalho do ambiente.
- **Mudanças de trabalho:** novos sistemas podem desqualificar os usuários ou fazê-los mudar a forma como trabalham.
- **Mudanças organizacionais:** o sistema pode alterar a estrutura do poder político em uma organização.

Os sistemas sociotécnicos definem que os processos operacionais e as pessoas que operam são partes relativas do sistema. Estes sistemas são regulados por políticas e regras organizacionais e são afetados por restrições e alterações externas, como leis e políticas nacionais de regulação (SOMMERVILLE, 2011).

Os sistemas sociotécnicos possuem três características importantes com relação à proteção e confiança:

1. Possuem propriedades emergentes que são do sistema como um todo e não apenas a partes individuais do sistema. As propriedades emergentes dependem dos componentes do sistema e dos relacionamentos entre eles e essa complexidade só pode ser avaliada uma vez, quando o sistema for montado. A proteção e a confiança fazem parte das propriedades emergentes do sistema.
2. Os sistemas sociotécnicos não são determinísticos, ou seja, quando apresentados a uma entrada específica, nem sempre produzem a mesma saída de dados. O seu comportamento vai depender de operadores humanos, e nem sempre as pessoas reagem da mesma maneira. Outra situação é com relação ao uso do sistema, que pode criar novos relacionamentos entre os componentes e com isso, alterar o seu comportamento emergente.
3. Os sistemas sociotécnicos apoiam os objetivos organizacionais, mas estes não dependem apenas do sistema. Eles podem depender da estabilidade, dos relacionamentos e conflitos entre os objetivos. Assim, um novo gerenciamento, pode, por exemplo, reinterpretar os objetivos organizacionais que o sistema suporta e de modo que um sistema “bem-sucedido” passa a ser visto como um sistema “fracasso”. As considerações sociotécnicas são importantes para determinar se um sistema está cumprindo com sucesso os seus objetivos.

A seguir, uma tabela (tabela 2) com exemplos de propriedades emergentes:

Tabela 2 – Exemplos de Propriedades Emergentes

PROPRIEDADE	DESCRIÇÃO
Volume	O volume de um sistema (o espaço total ocupado) varia conforme os conjuntos de componentes estão dispostos e conectados.
Confiabilidade	A confiabilidade de sistema depende da confiabilidade de componentes, mas interações inesperadas podem causar novos tipos de falhas e, portanto, afetar a confiabilidade do sistema.
Proteção	A proteção do sistema (sua capacidade de resistir ao ataque) é uma propriedade complexa que não pode ser facilmente mensurada. Os ataques podem ser criados de forma imprevista pelos projetistas de sistemas e, assim, derrotar as proteções internas.

PROPRIEDADE	DESCRIÇÃO
Reparabilidade	Essa propriedade reflete quanto fácil é corrigir um problema com o sistema uma vez que este tenha sido descoberto. Depende da capacidade de diagnosticar o problema e do acesso a componentes que estejam com defeito, bem como de se modificar ou substituir tais componentes.
Usabilidade	Essa propriedade reflete quanto fácil é usar o sistema. Depende dos componentes técnicos de sistema, seus operadores e seu ambiente operacional.

Fonte: Sommerville (2011, p. 188).



REFLITA

Pense seriamente sobre o software que você vai criar e pergunte a si mesmo “o que pode dar errado?”. Crie sua lista e peça a outros membros da equipe que façam o mesmo.

(Pressman e Maxim)

Falhas do Sistema

Nos sistemas críticos, quando ocorrem falhas, os custos são altíssimos e os erros costumam ser complexos, e muitas vezes, exigem pessoas experientes que assumam o controle, para contornar as dificuldades, e com isso as falhas serem corrigidas ou minimizadas.

Em sistemas críticos, existem três tipos de falhas que podem ocorrer:

- **Falhas de Hardware:** erros de fabricação, final de sua vida útil.
- **Falhas de Software:** enganos na especificação, projeto ou implementação.
- **Falhas Operacionais:** falha ao operar o sistema.

Vamos considerar a confiabilidade a partir de três perspectivas (figura 2) em sistemas sociotécnicos:

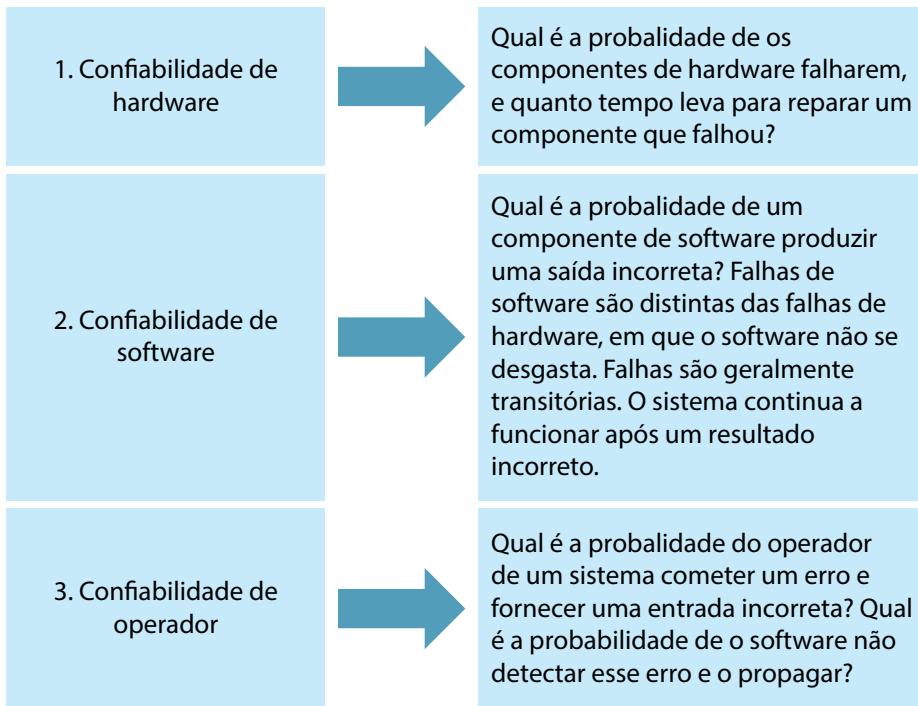


Figura 2 - Perspectivas de Confiabilidade em Sistemas Sociotécnicos

Fonte: adaptado de Sommerville, 2011, p. 189).

As falhas podem ser propagadas para os demais níveis do sistema (figura 3), pois a confiabilidade de hardware, de software e de operador não são independentes.

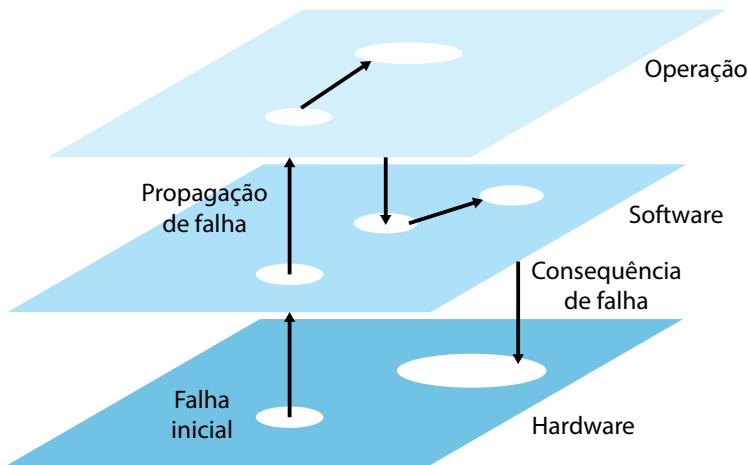


Figura 3 – Propagação da falha

Fonte: Sommerville (2011, p. 191).



REFLITA

Falha inicial, que poderia ser recuperável, pode rapidamente se transformar em um problema grave, que pode resultar em um desligamento completo do sistema.

(Sommerville)

Confiança e Proteção

Os sistemas computacionais fazem parte dos negócios e da vida pessoal das pessoas, e os problemas que resultam de falhas e defeitos nos softwares estão aumentando e, segundo Sommerville, sobre as falhas:

[...] uma falha do software de servidor em uma empresa de comércio eletrônico pode causar uma grande perda de receita e, inclusive, à perda dos clientes da empresa. Um erro de software em um sistema de controle embutido em um carro pode levar a recalls daquele modelo para reparação e, na pior das hipóteses, pode ser um fator de causa de acidentes. A infecção de PCs de uma empresa com malwares pode resultar na perda ou em danos a informações confidenciais, e requer operações de limpeza de alto custo para resolver o problema (SOMMERVILLE, 2011, 202).

Como os sistemas de software são tão importantes para os negócios das empresas e das pessoas, é essencial que ele seja confiável e que esteja disponível quando necessário e que funcione corretamente.

Mas o que é confiança de um sistema de computador? A confiança de um sistema de computador, segundo Sommerville (2011, p. 203):

[...] é uma propriedade do sistema que reflete sua fidedignidade. Fidedignidade aqui significa, essencialmente, o grau de confiança de um usuário no funcionamento esperado pelo sistema, no fato de que o sistema não ‘falhará’ em condições normais de uso.

A confiança não é expressa numericamente. Usamos termos relativos, como: ‘não confiável’, ‘muito confiável’ e ‘ultraconfiável’ para expressar o grau de confiança que podemos ter em um sistema de software.

E quando o usuário acha que um software não é confiável, mas ele é útil para determinada função? Muitas vezes, porque ser muito útil, o usuário está disposto a tolerar uma falha ocasional. E como lidar com as falhas? Como reflexo da desconfiança, os usuários acabam salvando seus trabalhos com maior frequência e manter múltiplas cópias de backup, entre outras estratégias. Assim, os usuários acabam compensando a falta de confiança dos sistemas por meio de ações que limitam os danos que poderiam resultar na falha de sistema.

Há quatro dimensões principais da confiança:

Tabela 3 - Principais Dimensões da confiança

Disponibilidade	Informalmente, a disponibilidade de um sistema é a probabilidade de que ele esteja pronto e em execução, capaz de fornecer serviços úteis a qualquer instante.
Confiabilidade	Informalmente, a confiabilidade de um sistema é a probabilidade, em um dado período de tempo, de que o sistema forneça corretamente os serviços, conforme esperado pelo usuário.
Segurança	Informalmente, a segurança de um sistema é um julgamento da probabilidade de que um sistema cause danos para pessoas ou para o ambiente.
Proteção	Informalmente, a proteção do sistema é um julgamento da probabilidade de que um sistema possa resistir a intrusões ou intencionais.

Fonte: adaptado de Sommerville (2011, p. 204).

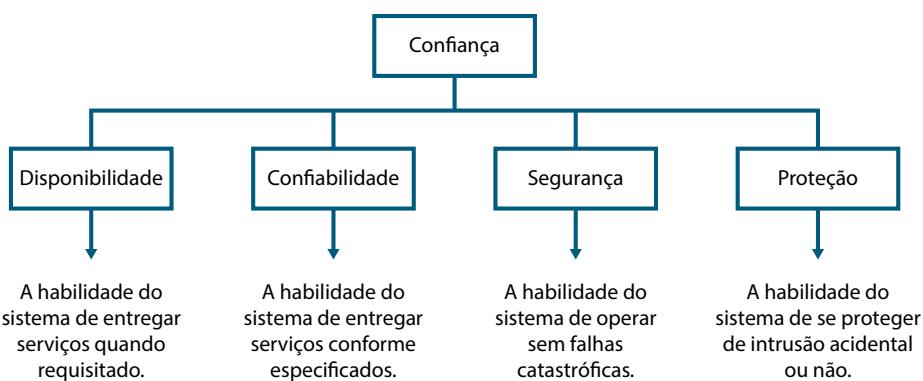


Figura 4 – Principais propriedades da confiança
Fonte: Sommerville (2011, p. 204).

A confiabilidade de um sistema vai depender do contexto em que ele for usado. Mas nem sempre o ambiente do sistema é completamente especificado e com isso, diferentes sistemas certos ambientes podem reagir a problemas de formas imprevisíveis, afetando a confiabilidade do sistema. Sobre a noção de confiança, Sommerville descreve que:

[...] a noção de confiança de sistema como uma propriedade abrangente foi desenvolvida porque as propriedades de confiança, disponibilidade, proteção, confiabilidade e segurança estão intimamente relacionadas. Geralmente, a operação segura de um sistema depende de ele estar disponível e operando de forma confiável. Um sistema pode tornar-se não confiável porque um intruso corrompeu seus dados. Ataques de negação de serviço em um sistema destinam-se a comprometer sua disponibilidade. Se um sistema é infectado por um vírus, sua confiabilidade e segurança ficam abaladas, pois o vírus pode mudar seu comportamento (SOMMERVILLE, 2011, p. 189).

Portanto, para desenvolver um software confiável, você precisa garantir que:

1. Seja evitada a introdução de erros acidentais no sistema durante a especificação e o desenvolvimento de software.
2. Sejam projetados processos de verificação e validação, eficazes na descoberta de erros residuais que afetam a confiança do sistema.
3. Sejam projetados mecanismos de proteção que protejam contra ataques externos capazes de comprometer a disponibilidade ou a proteção do sistema.
4. O sistema implantado e seu software de suporte sejam configurados corretamente para seu ambiente operacional. Além disso, você deve assumir que o software não é perfeito e que as falhas de software podem ocorrer. Seu sistema deve, portanto, incluir mecanismos de recuperação que tornem possível a restauração do serviço normal do sistema o mais rapidamente possível.



ENGENHARIA DE SEGURANÇA

Será que existe algum sistema de software que seja seguro? Muitos softwares que usamos enfrentam ameaças o tempo todo, desde aplicativos para internet até sistemas complexos. Engenheiros de software devem conhecer essas ameaças para que consigam ter a capacidade de proteger os sistemas desenvolvidos.

Segundo Pressman (2016), as ameaças existem há mais de uma década e estão se multiplicando com o crescimento da Web, a ubiquidade dos aplicativos móveis e o uso da nuvem. Para o autor, o uso dessas tecnologias vem gerando novas preocupações sobre a segurança com relação a privacidade do usuários, perdas ou roubo de informações pessoais e empresariais.

A segurança não preocupa apenas as pessoas que desenvolvem software para instituições militares, governamentais ou órgãos da saúde. Atualmente, a segurança deve ser uma preocupação de qualquer engenheiro de software que tenha recursos dos clientes para proteger. Na acepção mais simples, a segurança de software fornece os mecanismos que permitem a um sistema de software proteger seus ativos contra ataques. Ativos são recursos de sistema que tem valor para um ou mais envolvidos. Os ativos incluem informações de banco de dados, arquivos, programas, espaço de armazenamento no disco rígido, memória de sistema ou até mesmo capacidade de processador. Os ataques frequentemente tiram proveito de pontos fracos ou de vulnerabilidades do software que permitem acesso não autorizado a um sistema (PRESSMAN, 2016, p. 585).

A segurança é considerada um dos aspectos importantes da garantia da qualidade do software. E quanto mais relatos de erros, mais difícil fica considerar que o sistema tem qualidade ou que seja possível aumentar a qualidade. Da mesma forma, que fica difícil aumentar a segurança com relatos de vulnerabilidade no sistema. A segurança deve ser considerada desde o início do processo de desenvolvimento do software, incorporada nas fases iniciais do projeto, na implementação e verificadas durante os testes e na implantação.

ANÁLISE DOS REQUISITOS DE SEGURANÇA

Devem ser determinados junto com o cliente os requisitos de segurança, a fim de identificar os ativos que devem ser protegidos, e, em caso de perda desses ativos, qual o custo associado a cada um. Pressman (2016, p. 585) comenta que “o valor da perda de um ativo é conhecido como sua *exposição*”. E os valores da perda podem ser medidos em termos de tempo ou custo para recuperar ou recriar um ativo.

Durante a construção do software, é importante pensar em antecipar as condições ou ameaças que possam vir a causar uma perda que possam danificar o sistema ou torná-lo inacessível aos usuários. Esse processo de antecipação é chamado de Análise de Ameaça. Depois de identificados as ameaças e vulnerabilidades do sistema, é necessário que sejam criados controles para evitar estes ataques e mitigar seus danos e possíveis perdas.

Todavia conforme Pressman, talvez não seja possível desenvolver um sistema que se defenda de todas as ameaças imagináveis, por isso, é necessário que os usuários sejam estimulados a fazer backups de dados considerados críticos para a empresa e tentar garantir controles de privacidades.



SAIBA MAIS

A ubiquidade do acesso à Web e o advento da computação em nuvem permitem novas formas de colaborações de negócios. Compartilhar informações e proteger a confidencialidade é uma tarefa difícil. Tornar segura a computação de várias partes aumenta os riscos de comportamento egoísta, a não ser que todas tenham confiança de que ninguém pode tirar proveito dos sistemas. Essa situação enfatiza uma dimensão psicológica da credibilidade e da segurança dos sistemas que não pode ser resolvida apenas pela engenharia de software.

Fonte: Pressman (2016, p. 588).

ANÁLISE DA ENGENHARIA DE SEGURANÇA

Para a análise de segurança temos algumas tarefas que devem ser pensadas. Elas consideram os detalhes funcionais e não funcionais do sistema junto com as regras de negócio da empresa. As tarefas são:

- **Levantamento de Requisitos de Segurança:** esta tarefa utiliza as técnicas gerais de levantamento de requisitos e são aplicadas ao levantamento de requisitos de segurança. Os requisitos de segurança são os não funcionais e eles influenciam o projeto de arquitetura do sistema.
- **Modelagem de Segurança:** é a tarefa onde é feita uma descrição formal da política de segurança do sistema baseado nas informações do levantamento de requisitos de segurança. A política de segurança descreve os principais requisitos de segurança e contém regras que descrevem como a segurança será imposta durante a operação do sistema. A modelagem de segurança é uma referência valiosa sobre o sistema durante o seu ciclo de vida. O modelo de segurança pode ser representado por gráficos ou por textos. Independente de sua representação, ele precisa contemplar alguns itens, considerados principais, como:
 1. Objetivos da política de segurança.
 2. Requisitos da Interface Externa.
 3. Requisitos de Segurança do Software.
 4. Regras de Operação.
 5. Especificações descrevendo a correspondência modelo-sistema.

Em alguns sistemas, os modelos podem ser representados por máquinas de estado, onde cada estado deve conter informações sobre os aspectos mais relevantes sobre a segurança do sistema. Máquina de estado finito é definida por uma lista de estados de transição possíveis a partir de cada estado atual.

- **Projeto de Medidas:** tarefa onde as métricas e medidas de segurança precisam se encontrar na avaliação das propriedades:
 - **Confiabilidade:** funcionar sob condições hostis.
 - **Credibilidade:** o sistema não se comporta de forma mal-intencionada.
 - **Capacidade de sobrevivência:** continua a funcionar mesmo estando comprometido.

As métricas de segurança ajudam os desenvolvedores a se basear em medidas que avaliam até que ponto a confiabilidade das informações ou a integridade do sistema podem estar em risco. Segundo Pressman (2016, p. 591) “as melhores medidas são aquelas que estão prontamente disponíveis durante o desenvolvimento ou operação do software”.

- **Verificações de Exatidão:** tarefa que precisa ocorrer ao longo do ciclo de desenvolvimento do software. No início do processo de desenvolvimento devemos determinar qual a exposição de ativos envolvidos nos ataques contra vulnerabilidade do sistema. Muitas das verificações de segurança devem ser incluídas nas tarefas da Engenharia de Software convencional como nas auditorias, inspeções e atividades de teste.



Um software seguro deve exibir três propriedades: *confiabilidade, credibilidade e capacidade de sobrevivência*.

(Pressman)

ANÁLISE DE RISCO DE SEGURANÇA

Fazem parte do planejamento do projeto as tarefas de identificar e gerenciar riscos de segurança. A Engenharia de Segurança é orientada pelos riscos que são identificados pela equipe de software, pois os riscos impactam diretamente as atividades de gerenciamento e garantia da segurança do projeto (PRESSMAN, 2016).

Um conceito importante é a Modelagem de ameaças, que conforme Pressman (2016, pg. 594) “é um método de análise de segurança usado para identificar ameaças com o potencial mais alto de causar danos a um sistema baseado em software”. A modelagem de ameaças é realizada nas fases iniciais do projeto, usando as informações do levantamento de requisitos e os modelos de análise.

Para criar um modelo de ameaças é necessário:

- Identificar os componentes mais importantes de um sistema.
- Decompor o sistema.
- Identificar e classificar as possíveis ameaças aos componentes.
- Avaliar e classificar as ameaças para cada componente.
- Avaliar os componentes com base em sua classificação de risco.
- Desenvolver estratégias de mitigação de riscos.

A seguir uma tabela (tabela 4) com os passos exigidos para se construir um modelo de ameaça.

Tabela 4 - Passos para criar um modelo de ameaça

1. Identificar ativos	Listar toda informação sigilosa e propriedade intelectual: onde está armazenada, como está armazenada e quem tem acesso.
2. Criar uma visão geral da arquitetura	Escrever casos de uso do sistema e construir um modelo dos componentes do sistema.
3. Decompor o aplicativo	O objetivo é garantir que todos os dados enviados entre componentes do aplicativo sejam validados.
4. Identificar ameaças	Anotar todas as ameaças que podem comprometer ativos do sistema, usando métodos como árvores de ataque ou padrões de ataque; frequentemente, o processo envolve examinar ameaças à rede, configuração do sistema host e a aplicativos.
5. Documentar as ameaças	Cria um formulário de informação de riscos, detalhando como cada ameaça deve ser monitorada e mitigada.
6. Classificar as ameaças	A maioria dos projetos não tem recursos suficientes para tratar de todas as ameaças concebíveis; portanto, elas precisam ser classificadas usando seu impacto e probabilidade.

Fonte: Pressman (2016, p. 594).

Para Pressman (2016, p. 599), a preocupação da Engenharia de Segurança de software é com:

[...] a engenharia de segurança de software se preocupa com o desenvolvimento de software que proteja contra ameaças os ativos que gerencia. As ameaças podem envolver ataques que exploram vulnerabilidades do sistema para comprometer a confiabilidade, integridade ou disponibilidade de seus serviços e dados. A gestão de riscos à segurança se preocupa com a avaliação do impacto de possíveis ameaças e com a produção de requisitos de segurança para minimizar perdas críticas. O projeto voltado a segurança envolve a criação de uma arquitetura de sistemas que minimiza a introdução de vulnerabilidades conhecidas. Os engenheiros de software devem utilizar técnicas para evitar, repelir e recuperar de ataques, como uma maneira de mitigar os efeitos de perdas.

É necessário que os desenvolvedores considerem a garantia da segurança como uma atividade extensiva e que esteja presente desde o início do processo de desenvolvido do software e que acompanhe todo o ciclo de vida do sistema.

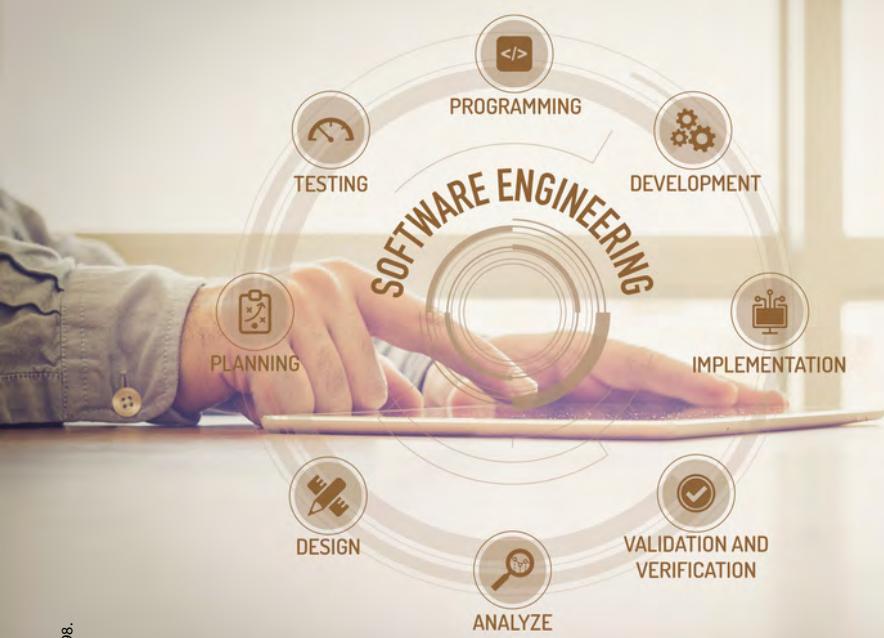


SAIBA MAIS

Sistemas de proteção

Um sistema de proteção é um sistema especializado, associado a algum outro sistema. Geralmente, é um sistema de controle de algum processo, como um processo de fabricação de produtos químicos ou um sistema de controle de equipamentos, como o sistema em um trem sem condutor. Um exemplo de um sistema de proteção pode ser um sistema em um trem que detecta se o trem passou por um sinal vermelho. Caso isso aconteça e não exista uma indicação de que o sistema de controle está desacelerando o trem. Então o sistema de proteção automaticamente aciona os freios do trem para levá-lo a uma parada. Os sistemas de proteção monitoram o ambiente de forma independente e, se os sensores indicarem algum problema com o qual o sistema controlado não esteja lidando, então o sistema de proteção é ativado para parar o processo ou o equipamento.

Fonte: Sommerville (2011, p. 242)



REENGENHARIA E MANUTENÇÃO DE SOFTWARE

Com tantas correções de bugs, solicitações de adaptações e melhorias, que devemos pensar em enfrentar o desafio da manutenção do software, pois a fila cresceu e a empresa está gastando mais tempo e dinheiro para a manutenção do sistema do que com o desenvolvimento de novas aplicações. Outro problema é mobilidade dos profissionais, pois é bem provável que os desenvolvedores pelo código original não esteja mais na empresa ou outros desenvolvedores tenham modificado o sistema e já se foram ou pode não ter restado ninguém na empresa que tenha conhecimento direto do sistema legado (PRESSMAN, 2016, p. 797).

A evolução do sistema envolve a compreensão e o conhecimento do programa que tem que ser mudado, para a implementação de novas mudanças. Todavia muitos sistemas legados são velhos, difíceis de serem mudados. Ao longo do tempo, a estrutura inicialmente projetada do sistema pode não ser mais suportada ou pode ter sido danificada por várias mudanças durante o seu ciclo de vida. E para que os sistemas legados sejam mais fáceis para serem mantidos, Sommerville fala que é preciso melhorar a sua estrutura e inteligibilidade aplicando a reengenharia.

A reengenharia pode envolver a redocumentação de sistema, a refatoração da arquitetura de sistema, a mudança de linguagem de programação para uma linguagem moderna e modificações e atualizações da estrutura e dos dados de sistema. A funcionalidade de software não é alterada, e você geralmente deve evitar grandes mudanças na arquitetura de sistema (SOMMERVILLE, 2011, p. 174).

A seguir os dois benefícios (tabela 5) que são considerados importantes na reengenharia:

Tabela 5 – Benefícios importantes da Reengenharia

1. Risco reduzido	Existe um alto risco em desenvolver novamente um software crítico de negócios. Podem ocorrer erros na especificação de sistema ou pode haver problemas de desenvolvimento. Atrasos no início do novo software podem significar a perda do negócio e custos adicionais.
2. Custo reduzido	O custo de reengenharia pode ser significativamente menor do que o de desenvolvimento de um novo software.

Fonte: Sommerville (2011, p. 174).

Abaixo temos um modelo geral de processo de reengenharia (figura 5), onde a entrada para o processo é um programa legado (programa original) e a saída, uma versão melhorada e reestruturada do mesmo programa (dados reconstruídos).

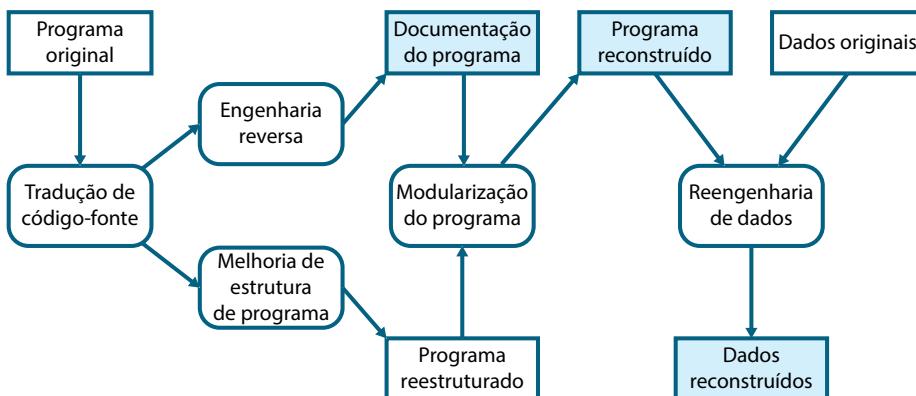


Figura 5 - O processo de reengenharia

Fonte: Sommerville (2011, p. 175).

O processo de reengenharia tem as seguintes atividades:

- 1. Tradução de código-fonte:** atividade onde é utilizada uma ferramenta de tradução. O programa é convertido a partir de uma linguagem de programação antiga para uma versão mais moderna da mesma linguagem (se houver) ou em outra diferente.
- 2. Engenharia reversa:** atividade onde o programa é analisado para extrair informações sobre ele, para que seja documentado a sua organização e as funcionalidades. Esse processo pode ser automatizado.

- 3. Melhoria de estrutura de programa:** atividade onde a estrutura de controle do programa é analisada e pode ser modificada caso seja necessário, para que se torne mais fácil de ler e entender. Esse processo pode ser parcialmente automatizado, pois em algum momento, pode ser exigida alguma intervenção manual.
- 4. Modularização de programa:** atividade onde as partes relacionadas do programa são agrupadas, e caso haja redundância, esta poderá ser removida. Esse é um processo manual.
- 5. Reengenharia de dados:** atividade em que os dados processados pelo programa são alterados para que reflitam as mudanças do programa. Esse processo pode significar a redefinição dos esquemas de banco de dados e a conversão do banco de dados existente para a nova estrutura. Exigindo que os dados sejam limpos, que erros sejam encontrados e corrigidos, que sejam removidos registros duplicados etc. A reengenharia de dados só se torna necessária se as estruturas de dados de programa mudar durante o processo da reengenharia de sistema.

Dependendo da extensão do trabalho, os custos da reengenharia podem aumentar. Como mostra a figura 6, temos um espectro de possíveis abordagens para a reengenharia, onde os custos aumentam da esquerda para a direita, de modo que a conversão de código-fonte automatizada é a opção mais barata e a reestruturação mais as mudanças de arquitetura é a mais cara.

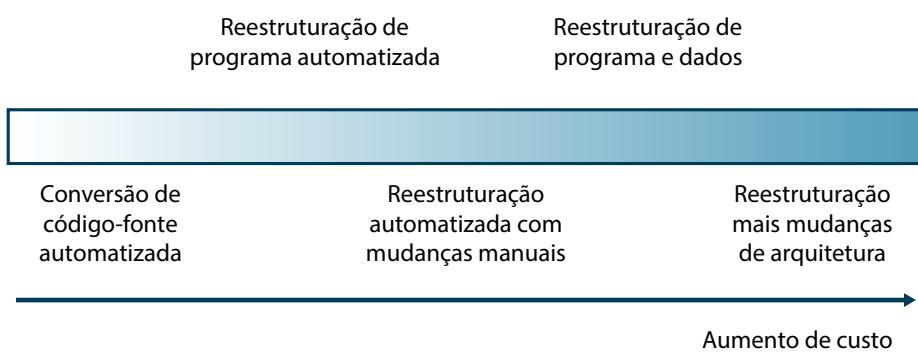


Figura 6 - Abordagens da reengenharia
Fonte: (Sommerville, 2011, p.176).

Sobre esse aumento de custo, Sommerville (2011, pg. 175) fala que “o problema com a reengenharia de software é que existem limites práticos para o quanto você

pode melhorar um sistema por meio da reengenharia". Por exemplo, não é possível converter um sistema desenvolvido com uma abordagem funcional para um sistema desenvolvido com a abordagem orientado a objetos. Assim, principais mudanças que envolvem a arquitetura ou a reorganização radical do sistema de gerenciamento de dados, por exemplo, não podem ser feitas automaticamente, pois são mudanças muito caras.

Atividades de Reengenharia de Software

O modelo de processo de reengenharia de software mostrado na Figura 7 é um modelo cíclico, ou seja, cada uma das atividades apresentadas como parte do modelo pode ser revisitada e para qualquer ciclo, o processo pode terminar após qualquer uma dessas atividades (SOMMERVILLE, 2016).

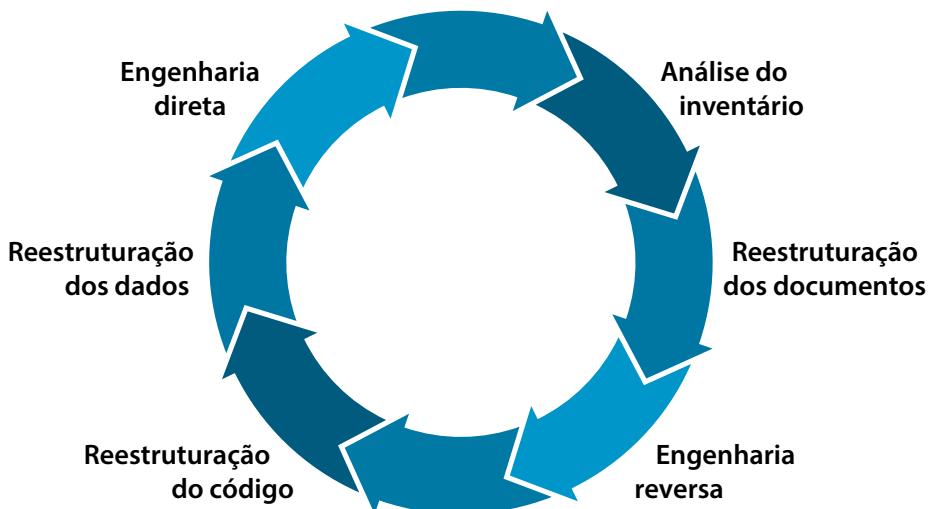


Figura 7 – Um modelo de processo de reengenharia de software

Fonte: (Pressman, 2016, p. 803).

- **Análise de inventário:** toda empresa de software deve ter um inventário de todos os aplicativos. O inventário pode ser: planilhas com informações detalhadas e ordenadas (tamanho, idade, criticalidade nos negócios, longevidade, manutenibilidade atual e suportabilidade) para cada aplicação ativa.

- **Reestruturação dos documentos:** documentação pobre é a marca registrada de muitos sistemas legados. Quando o sistema não tem nenhuma documentação, é muito dispendioso tentar criar uma. Em alguns casos, crie documentação somente daquilo que sofreu alterações. Crie apenas a documentação necessária para que seja possível entender o software.
- **Engenharia reversa:** é o processo para analisar um programa na tentativa de criar uma representação do sistema em um nível mais alto de abstração do que o código-fonte. Logo mais, vamos ver com mais detalhes este item.
- **Reestruturação do código:** é o tipo mais comum de reengenharia. Por exemplo, em alguns sistemas legados, a arquitetura é razoavelmente sólida e os módulos individuais foram codificados de um modo que se torna difícil de entendê-los, testá-los e mantê-los, mas os códigos dentro dos módulos podem ser reestruturados.
- **Reestruturação dos dados:** um sistema com uma arquitetura de dados fraca é difícil de adaptar e melhorar. Na reestruturação de código ocorre um nível relativamente baixo de abstração, e a reestruturação de dados é considerada uma atividade de reengenharia completa.
- **Engenharia direta:** recupera as informações do projeto do software existente e também usa as informações para alterar ou reconstituir o sistema existente em um esforço para melhorar sua qualidade geral.

A reengenharia pode melhorar a manutenibilidade do sistema, mas o sistema reconstruído provavelmente não será de fácil manutenção como um novo sistema, desenvolvido com métodos modernos de engenharia de software (SOMMERVILLE, 2011).

MANUTENÇÃO DE SOFTWARE

O software continuará evoluindo com o passar do tempo, independentemente do domínio de aplicação, tamanho ou da sua complexidade. E esse processo de evolução é dirigido pelas mudanças que ocorrem no software quando ocorrem alterações quando erros são corrigidos, quando há adaptação a um novo ambiente ou tecnologia, solicitações de novas funcionalidades pelo cliente, novas regras de negócio exigidas pelo governo e quando o sistema passa por um processo de reengenharia.

Sobre a manutenção de software Pressman descreve que ela começa quase imediatamente.

A manutenção começa quase imediatamente. O software é liberado para os usuários finais, e em alguns dias, os relatos de bugs começam a chegar à organização de engenharia de software. Em algumas semanas, uma classe de usuários indica que o software deve ser mudado para se adaptar às necessidades especiais de seus ambientes. E em alguns meses, outro grupo corporativo, ainda não interessado no software quando foi lançado, agora reconhece que pode lhes trazer alguns benefícios. Eles precisarão de algumas melhorias para fazer o software funcionar em seu mundo (PRESSMAN, 2016, p. 796).

Será que é necessário a manutenção e todo esforço despendido para realizá-la? Sim, e uma das razões é que muitos softwares que usamos hoje, e dependemos, possuem em média de 10 a 15 anos. Esses sistemas, quando foram desenvolvidos, usaram na época as melhores técnicas de projeto e codificação conhecidas. O tamanho do sistema e o espaço de armazenamento eram as preocupações principais nesta época. E hoje, muitos desses sistemas migraram para novas plataformas, ajustados nas novas tecnologias e aperfeiçoados para atender a novas necessidades dos usuários. E apesar de todas as mudanças, a arquitetura geral do sistema não teve uma grande atenção, e o resultado são estruturas mal projetadas, mal codificadas, de lógica pobre e mal documentadas e que exigem muitos chamados para corrigir falhas e inconsistência a fim de mantê-los rodando (PRESSMAN, 2016, 797).

Para Pressman (2016, p. 797) “outra razão para o problema de manutenção do software é a mobilidade dos profissionais”. Possivelmente, a equipe de desenvolvedores original do sistema não seja mais a mesma ou outras gerações de desenvolvedores podem ter modificado o sistema e já se foram. E hoje na empresa, pode não ter restado nenhum desenvolvedor que tenha conhecimento do sistema legado.

As mudanças são inevitáveis, portanto, devem-se desenvolver mecanismos para avaliar, controlar e fazer modificações ao longo do ciclo de vida do software. Em algumas disciplinas de Engenharia de Software, foi destacada a importância de entender o problema do cliente na fase de análise e desenvolver uma solução bem estruturada na fase de projeto. Tanto a fase de análise

quanto a fase de projeto levam a uma importante característica do software que chamamos de manutenibilidade, que facilita com que o software pode ser corrigido, adaptado ou melhorado ao longo do seu ciclo de vida. Muitas funções da Engenharia de Software são usadas para criar sistemas que apresentem alta manutenibilidade (PRESSMAN, 2016).

Sobre a manutenção de software Sommerville descreve que:

[...] a manutenção de software é o processo geral de mudança em um sistema depois que ele é liberado para uso. O termo geralmente se aplica ao software customizado em que grupos de desenvolvimento separados estão envolvidos antes e depois da liberação. As alterações feitas no software podem ser simples mudanças para correção de erros de codificação, até mudanças mais extensas para correção de erros de projeto, ou melhorias significativas para corrigir erros de especificação ou acomodar novos requisitos. As mudanças são implementadas por meio da modificação de componentes do sistema existente e, quando necessário, por meio da adição de novos componentes (SOMMERVILLE, 2011, p. 172).

Existem três diferentes tipos de Manutenção de Software, conforme a tabela abaixo:

Tabela 6 – Tipos de Manutenção de Software

TIPOS DE MANUTENÇÃO DE SOFTWARE	
Correção de defeitos	Erros de codificação são relativamente baratos para serem corrigidos; erros de projeto são mais caros, pois podem implicar em reescrever vários componentes de programa. Erros de requisitos são os mais caros para se corrigir devido ao extenso reprojeto de sistema que pode ser necessário.
Adaptação ambiental	Esse tipo de manutenção é necessário quando algum aspecto do ambiente do sistema, como o hardware, a plataforma do sistema operacional ou outro software de apoio sofre uma mudança. O sistema de aplicação deve ser modificado para se adaptar a essas mudanças de ambiente.
Adição de funcionalidade	Esse tipo de manutenção é necessário quando os requisitos de sistema mudam em resposta às mudanças organizacionais ou de negócios. A escala de mudanças necessárias para o software é, frequentemente, muito maior do que para os outros tipos de manutenção.

Fonte: adaptado de Sommerville (2011, p. 172).

Não existe uma distinção clara entre esses tipos de manutenção na prática. Os tipos de manutenção geralmente são reconhecidos, mas costumam dar-lhes nomes diferentes. A manutenção corretiva é universalmente usado para se referir à manutenção para corrigir defeitos. Todavia a manutenção adaptativa significa adaptação ao novo ambiente e pode significar, em outro momento, a adaptação do software aos novos requisitos. Já a manutenção perfectiva significa aperfeiçoar o software por meio da implementação de novos requisitos e em outros casos, significa a manutenção de funcionalidade de sistema para melhorar sua estrutura e seu desempenho (SOMMERVILLE, 2011).

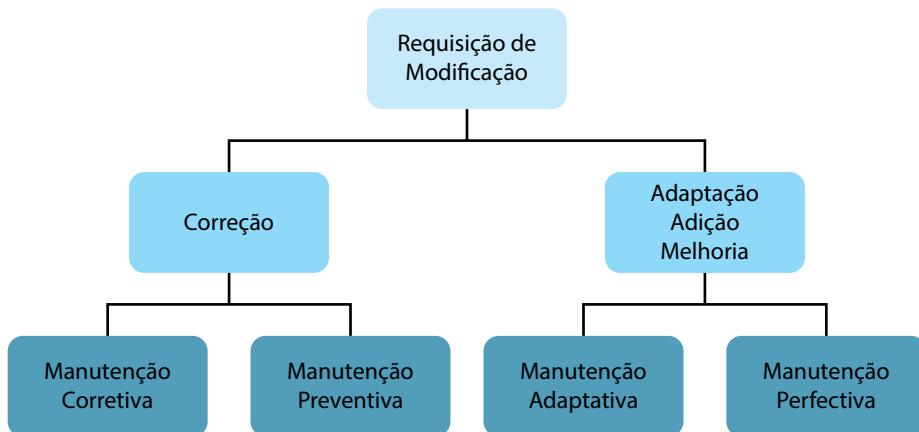


Figura 8 - Tipos de Manutenção

Fonte: a autora.

Entretanto, caro(a) aluno (a), você deve estar se perguntando: o que é manutenibilidade? Manutenibilidade é o software “manutenível” que apresenta uma modularidade eficaz, faz uso de padrões de projeto que possibilitem entendê-lo facilmente, além de ser construído usando padrões e convenções de codificação que sejam bem definidos e que levam a um código-fonte auto documentado e inteligível. Que tenha passado por uma variedade de técnicas de garantia de qualidade e que se descobriram potenciais problemas de manutenção antes que o software fosse lançado e que foi criado por desenvolvedores que sabem que não estarão por perto quando alterações e mudanças tiverem que ser feitas. Ou seja, que as fases do projeto e da implementação do software ajudem a quem for fazer as alterações no futuro.

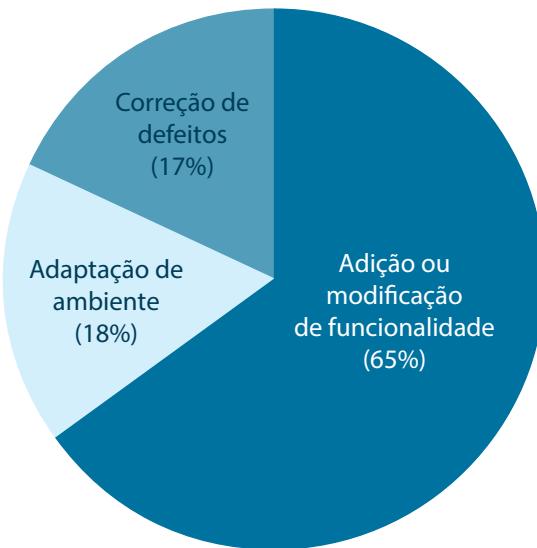


Figura 9 – Distribuição do esforço de manutenção

Fonte: Sommerville (2011, p. 171).

Nas fases de projeto e implementação de software, é importante pensar em investir esforços para a redução de custos de mudanças futuras. Sobre a distribuição do esforço e da manutenção (figura 9) Sommerville fala que:

[...] adicionar uma nova funcionalidade após a liberação é caro porque é necessário tempo para aprender o sistema e analisar o impacto das alterações propostas. Portanto, o trabalho feito durante o desenvolvimento para tornar a compreensão e a mudança no software mais fáceis provavelmente reduzirá os custos de evolução. Boas técnicas de engenharia de software, como descrição precisa, uso de desenvolvimento orientado a objetos e gerenciamento de configuração, contribuem para a redução dos custos de manutenção (SOMMERVILLE, 2011, pg. 171)

Conforme o autor, é mais caro adicionar funcionalidade depois que um software está em uso do que implementar a mesma funcionalidade durante o seu desenvolvimento e as razões para isso são:

- 1. Estabilidade da equipe:** após o sistema ter sido liberado para uso do cliente, muitas vezes a equipe de desenvolvimento é desmobilizada e remanejada para novos projetos. E muitas vezes, a equipe ou as pessoas responsáveis pela manutenção deste sistema não entendem o sistema ou não entendem a regra de negócios do cliente.

2. **Más práticas de desenvolvimento:** em muitos casos, o contrato para a manutenção de um sistema é separado do contrato de desenvolvimento do sistema. E pode ocorrer do contrato de manutenção ser dado a uma empresa diferente da do desenvolvedor do sistema original.
3. **Qualificações de pessoal:** em geral, a equipe de manutenção é inexperiente e não domina a regra de negócio da aplicação. Segundo Sommerville (2011, p. 172), “a manutenção tem uma imagem pobre entre os engenheiros de software. É vista como um processo menos qualificado do que o desenvolvimento de sistema e é muitas vezes atribuída ao pessoal mais jovem”. Em alguns casos, os sistemas antigos podem ter sido desenvolvidos em linguagens obsoletas de programação e a equipe de manutenção pode não ter muita experiência nessas linguagens.
4. **Idade do programa e estrutura:** muitas vezes, o sistema sofre muitas alterações e com isso a sua estrutura tende a degradar e envelhecer, tornando-se cada vez mais difíceis de serem entendidos e alterados. Em alguns casos, os sistemas foram desenvolvidos sem técnicas modernas de engenharia de software, as documentações podem ter se perdido ou ser inconsistentes. E muitas vezes, não são submetidos a um gerenciamento de configuração, e com isso, acaba desperdiçando muito tempo para encontrar as versões corretas dos componentes do sistema para a alteração.

O uso de técnicas de reengenharia de software pode ajudar a melhorar a estrutura do sistema e sua inteligibilidade, e as transformações de arquitetura podem adaptar o sistema para um novo hardware. Outra técnica que pode ajudar é a refatoração que pode melhorar a qualidade do código do sistema e facilitar a mudança.



REFLITA

Reengenharia dos sistemas de informação é uma atividade que absorverá recursos da tecnologia de informação por muitos anos.

(Pressman)



ENGENHARIA REVERSA

Sobre o termo Engenharia Reversa e suas origens, Pressman (2016, p. 805) descreve que:

O termo *engenharia reversa* tem suas origens no mundo do hardware. Uma empresa desmonta um produto de hardware competitivo na tentativa de conhecer os “segredos” de projeto e fabricação do concorrente. Os segredos poderiam ser facilmente entendidos se fosse possível obter as especificações de projeto e fabricação do concorrente. Mas esses documentos são de propriedade privada e não estão disponíveis para a empresa que está fazendo a engenharia reversa. Essencialmente, uma engenharia reversa bem-sucedida resulta em uma ou mais especificações de projeto e fabricação para um produto pelo exame de amostras atuais do produto. A engenharia reversa para o software é bem similar.

Na engenharia reversa para o software, no entanto, o sistema a ser submetido a uma engenharia reversa não é o sistema de um concorrente. Ela é aplicada no próprio sistema da empresa e em geral, foi desenvolvimento há muitos anos. O conceito de Engenharia Reversa para o software, segundo Pressman (2016, p. 805), é “o processo para analisar um programa na tentativa de criar uma representação do programa em um nível mais alto de abstração do que o código-fonte”.

Podemos falar que a engenharia reversa é um processo de recuperação do projeto, onde as ferramentas extraem informações do projeto de dados, da arquitetura e procedural com base em um programa já desenvolvido. As informações podem ser extraídas do projeto do código-fonte pela engenharia reversa.

Tabela 7 – Informações extraídas da Engenharia Reversa

Nível de abstração

O nível de abstração de um processo de engenharia reversa e as ferramentas utilizadas para realizá-lo referem-se à sofisticação das informações de projeto que podem ser extraídas do código-fonte. Idealmente, o nível de abstração deverá ser tão alto quanto possível. O processo de engenharia reversa deverá ser capaz de derivar representações de projeto procedural (em uma abstração de baixo nível), informações de programa e de estrutura de dados (em um nível de abstração um tanto mais alto), modelos de objeto, dados e/ou modelos de fluxo de controle (em um nível de abstração relativamente alto) e modelos de entidade-relacionamento (em um nível alto de abstração).

Completeza

A completeza de um processo de engenharia reversa refere-se ao nível de detalhe fornecido em um nível de abstração. Em muitos casos, a completeza diminui conforme o nível de abstração aumenta. A completeza melhora em proporção direta com a quantidade de análise executada pela pessoa que está fazendo a engenharia reversa.

Interatividade

A interatividade refere-se ao grau segundo o qual as pessoas são “integradas” com as ferramentas automáticas para criar um processo eficaz de engenharia reversa. Em muitos casos, na medida em que o nível de abstração aumenta, a interatividade deve aumentar ou a completeza será prejudicada.

Direcionalidade

Se a direcionalidade do processo de engenharia reversa for um único sentido, todas as informações extraídas do código-fonte serão fornecidas ao engenheiro de software que pode então usá-las durante qualquer atividade de manutenção. Se a direcionalidade for nos dois sentidos, as informações serão colocadas em uma ferramenta de reengenharia que tenta reestruturar ou regenerar o sistema antigo.

Fonte: Pressman (2016, p. 805).

Conforme Pressman (2016, p. 806), o processo de engenharia reversa, representado na figura 10, tem a seguinte descrição:

[...] antes de começar as atividades de engenharia reversa, o código-fonte não estruturado (“ruim”) é reestruturado para que contenha somente as construções de programação estruturada. Isso torna o código-fonte mais fácil de ler e proporciona a base para todas as atividades subsequentes de engenharia reversa. O núcleo da engenharia reversa é uma atividade chamada de *extração de abstrações*. Você deve avaliar o programa antigo e, com base no código-fonte (muitas vezes não documentado), desenvolver uma especificação significativa do processamento executado, da interface de usuário aplicada e das estruturas de dados de programas ou do banco de dados utilizadas.

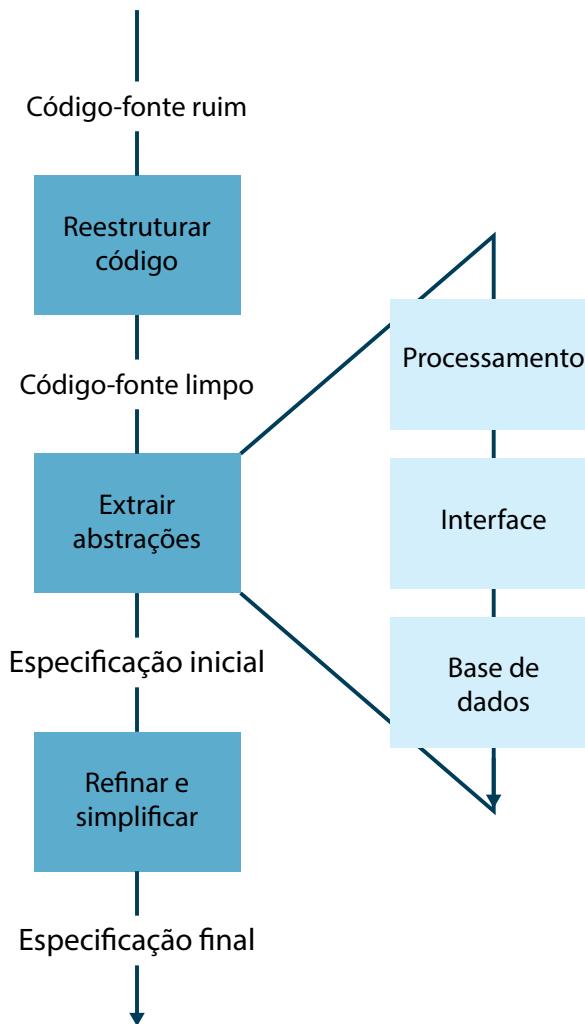


Figura 10 – O processo de Engenharia Reversa

Fonte: Pressman (2016, p. 806).

REFLITA



Três problemas de Engenharia Reversa devem ser tratados: nível de abstração, completude e direcionalidade.

(Pressman)

ENGENHARIA REVERSA PARA ENTENDER OS DADOS

Para entendermos a engenharia reversa dos dados, vamos conhecer os níveis onde ela ocorre:

- **Nível de abstração:** a engenharia reversa dos dados ocorre em diferentes níveis de abstração e normalmente é a primeira tarefa da reengenharia.
- **Nível de programa:** as estruturas internas de dados de programa devem passar por uma engenharia reversa como parte de um trabalho de reengenharia total.
- **Nível de sistema:** as estruturas de dados globais (por exemplo, arquivos, bases de dados) passam por uma reengenharia para acomodar novos paradigmas de gerenciamento de base de dados.

Segundo Pressman (2011, p. 807), “a engenharia reversa das estruturas de dados globais atuais define o cenário para a introdução de uma nova base de dados para todo o sistema”.

Nas **estruturas internas de dados** às técnicas de engenharia reversa para dados internos de programa destacam a definição de classes de objetos. O código do programa é examinado para agrupar variáveis de programa relacionadas. A organização de dados dentro do código pode identificar tipos de dados abstratos, como por exemplo, estruturas de registro, arquivos, listas e outras estruturas de dados muitas vezes fornecem um indicador inicial das classes (PRESSMAN, 2016).

A **estrutura de base de dados** permite a definição de objetos de dados e o uso de algum método que estabeleça relações entre esses objetos, independentemente de sua organização lógica e estrutura física. É necessário entender os objetos e suas relações para fazer a reengenharia de um esquema de base de dados para outro (PRESSMAN, 2016).

ENGENHARIA REVERSA PARA ENTENDER O PROCESSAMENTO

Conforme Pressman (2016, p.807), “a engenharia reversa para entender o processamento começa com uma tentativa de entender e extrair abstrações procedurais representadas pelo código-fonte”. E para entender as abstrações procedurais é analisado o código-fonte nos seguintes níveis de abstrações:

- Sistema
- Programa
- Componente
- Padrão
- Instruções

Antes do trabalho detalhado na engenharia reversa, a funcionalidade global de todo o sistema deve ser entendida para estabelecer um contexto para uma análise posterior e também para proporcionar uma visão sobre os problemas de interoperabilidade entre os aplicativos no sistema. Na visão do Pressman (2016, p. 807) temos que:

[...] cada um dos programas que formam o sistema de um aplicativo representa uma abstração funcional em um alto nível de detalhe. É criado um diagrama de blocos, representando a interação entre essas abstrações funcionais. Cada componente executa alguma subfunção e representa uma abstração procedural definida. É desenvolvida uma narrativa de processamento para cada componente. Em algumas situações, já existem especificações de sistema, programa e componente. Quando esse é o caso, as especificações são revistas para verificar a conformidade com o código existente.

Ainda conforme o autor, a engenharia reversa para sistemas grandes em geral é feita usando-se uma abordagem semiautomática. Podem ser empregadas ferramentas automatizadas para ajudá-lo a entender a semântica do código existente. O resultado desse processo é então passado para as ferramentas de reestruturação e engenharia direta para completar o processo de reengenharia (PRESSMAN, 2016). Na Leitura Complementar temos os conceitos básicos sobre a reestruturação e engenharia direta.

ENGENHARIA REVERSA DAS INTERFACES DE USUÁRIO

As interfaces de usuários sofisticadas tornaram-se uma exigência para sistemas de todos os tipos baseados em computadores, e por isso, tornou-se uma das atividades mais comuns de reengenharia. Sobre isso Pressman (2016, p. 808) define que:

[...] antes de recriar uma interface de usuário, deverá ocorrer a engenharia reversa. Para entender completamente uma interface de usuário, deve ser especificada a estrutura e o comportamento da interface. Grande parte das informações necessárias para criar um modelo comportamental pode ser obtida observando-se a manifestação externa da interface. Mas as informações adicionais necessárias para criar o modelo comportamental devem ser extraídas do código.

É importante notar que uma interface substituta pode não refletir exatamente a interface antiga (na verdade, pode ser radicalmente diferente). Muitas vezes compensa desenvolver uma nova metáfora de interação. Por exemplo, uma interface de usuário antiga requer que o usuário forneça um fator de escala (variando de 1 a 10) para reduzir ou ampliar uma imagem gráfica. Uma interface que passou pela reengenharia pode usar uma barra de rolagem e o mouse para executar a mesma função.

Na figura 11, a seguir, demonstra como são a Engenharia Reversa, Reengenharia e Engenharia Direta.

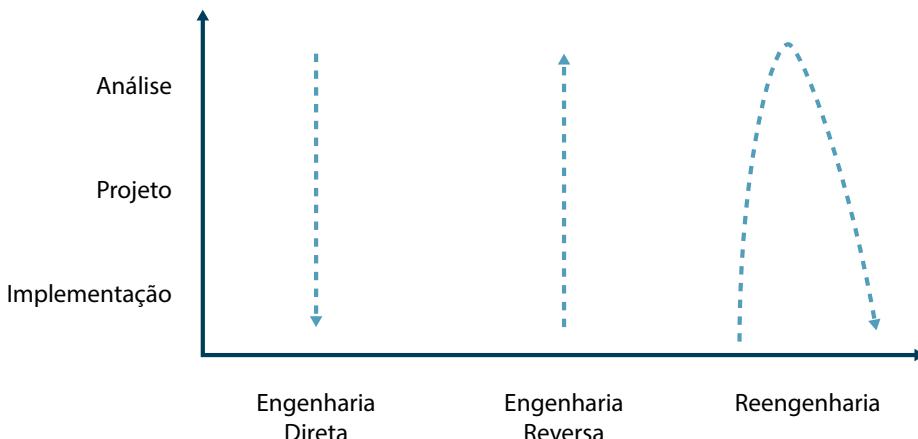


Figura 11 – Demonstrativo entre as Engenharias
Fonte: a autora.

A manutenção e o suporte de software são atividades contínuas que ocorrem por todo o ciclo de vida de um aplicativo. Durante essas atividades, defeitos são corrigidos, aplicativos são adaptados a um ambiente operacional ou de negócio em mutação, melhorias são implementadas por solicitação dos interessados e é fornecido suporte aos usuários quando integram um aplicativo em seu fluxo de trabalho pessoal ou corporativo. A reengenharia ocorre em dois níveis de abstração. No nível de negócio, ela concentra-se nos processos de negócio para melhorar a competitividade em alguma área do negócio. No nível de software, a reengenharia examina os sistemas de informação e os aplicativos, com a finalidade de reestruturá-los para que tenham uma melhor qualidade. A reengenharia de software abrange uma série de atividades que incluem análise de inventário, reestruturação da documentação, engenharia reversa, reestruturação de programas e dados e engenharia direta. (PRESSMAN, 2016)



CONSIDERAÇÕES FINAIS

Prezado aluno(a)! Chegamos ao final de mais uma unidade. Aprendemos conceitos sobre os Sistemas Críticos, Engenharia de Segurança, Reengenharia e Manutenção e Engenharia Reversa.

Aprendemos conceitos básicos sobre os Sistemas Críticos, seus tipos e suas propriedades como a confiança, considerada uma das mais importantes. Junto com sistemas críticos abordamos os conceitos de um sistema sociotécnico – que inclui pessoas, software e hardware – e aprendemos sobre o que é necessário para ter uma perspectiva de proteção e uma confiança no software. Outros conceitos que falamos foi sobre as falhas, a confiança e a proteção de software e como elas não devem ser tratadas isoladamente e como elas podem afetar drasticamente um sistema.

Estudamos os conceitos que envolve a Engenharia de Segurança. Vimos como a segurança é considerada um dos aspectos mais importantes da garantia da qualidade do software. Aprendemos sobre a Reengenharia e a Manutenção de Software. E vimos como a evolução do sistema envolve a compreensão e o conhecimento do programa que tem que ser alterado, para a implementação de novas mudanças. A reengenharia pode melhorar a manutenibilidade do sistema, mas o sistema reconstruído provavelmente não será de fácil manutenção como um novo sistema. O software continuará evoluindo com o passar do tempo, independentemente do domínio de aplicação, tamanho ou da sua complexidade.

Outro conceito importante que vimos nesta unidade foi sobre Engenharia Reversa. Aprendemos que a engenharia reversa é um processo de recuperação do projeto, onde as ferramentas extraem informações do projeto de dados, da arquitetura e procedural com base em um programa já desenvolvido. As informações podem ser extraídas do projeto do código-fonte pela engenharia reversa.

Depois desta Unidade, com o conhecimento que já adquirimos podemos passar para a próxima Unidade, para que neste modo você se aprofunde ainda mais no conhecimento. Preparados? Então vamos em frente! Bons estudos!

ATIVIDADES



1. Segundo Pressman (2016), a manutenção e o suporte de software são atividades contínuas que ocorrem por todo o ciclo de vida de um aplicativo. Durante essas atividades, defeitos são corrigidos, aplicativos são adaptados a um ambiente operacional ou de negócio em mutação, melhorias são implementadas por solicitação dos interessados e é fornecido suporte aos usuários quando integram um aplicativo em seu fluxo de trabalho pessoal ou corporativo. Com base nessa informação, analise as alternativas abaixo:

- I. O software continuará evoluindo com o passar do tempo, independentemente do domínio de aplicação, tamanho ou da sua complexidade.
- II. A equipe de desenvolvedores original do sistema tem que ser a mesma para executar as mudanças no sistema.
- III. As mudanças são inevitáveis, portanto, devem-se desenvolver mecanismos para avaliar, controlar e fazer modificações ao longo do ciclo de vida do software.
- IV. A manutenção de software é o processo geral de mudança em um sistema antes que ele seja liberado para uso.

É correto o que se afirma em:

- a) I, II apenas.
- b) I, III apenas.
- c) I, II, III apenas.
- d) I, III, IV apenas.
- e) I, II, III, IV apenas.

ATIVIDADES



2. A engenharia reversa evoca uma imagem da “fenda mágica”. Você coloca na fenda uma listagem de código não documentada, criada de qualquer jeito, e do outro lado sai uma descrição completa de projeto (e documentação completa) para o programa do computador. Infelizmente, a fenda mágica não existe. A engenharia reversa pode extrair informações do projeto do código-fonte, mas o nível de abstração, a completeza da documentação, o grau segundo o qual as ferramentas e um analista humano trabalham juntos e a direcionalidade do processo são altamente variáveis (PRESSMAN, 2016).

- I. Com base nessa informação, analise as alternativas abaixo:
- II. A completeza de um processo de engenharia reversa refere-se ao nível de detalhe fornecido em um nível de abstração.
- III. A completeza diminui conforme o nível de abstração aumenta.
- IV. Em muitos casos, na medida em que o nível de abstração aumenta, a interatividade deve aumentar ou a completeza será prejudicada.
- V. A completeza melhora em proporção direta com a quantidade de análise executada pela pessoa que está fazendo a engenharia reversa

É correto o que se afirma em:

- a) I, II apenas.
- b) I, III apenas.
- c) I, II, III apenas.
- d) I, III, IV apenas.
- e) I, II, III, IV apenas.

ATIVIDADES



3. Segundo Pressman (2016), a reengenharia de software abrange uma série de atividades que incluem análise de inventário, reestruturação da documentação, engenharia reversa, reestruturação de programas e dados e engenharia direta. Pensando nisso, relacione as colunas:

- | | |
|-----------------------|--|
| 1. Nível de abstração | () É um processo de engenharia reversa, refere-se ao nível de detalhe fornecido em um nível de abstração. |
| 2. Completeza | () Nível de processo de engenharia reversa e as ferramentas utilizadas para realizá-lo, referem-se à sofisticação das informações de projeto que podem ser extraídas do código-fonte. |
| 3. Interatividade | () Refere-se ao grau segundo o qual as pessoas são “integradas” com as ferramentas automáticas para criar um processo eficaz de engenharia reversa. |
- a) 1, 2, 3.
b) 1, 3, 2.
c) 3, 2, 1.
d) 2, 1, 3.
e) 3, 1, 2.

ATIVIDADES



4. Em um sistema de computador, o software e o hardware são interdependentes. Sem hardware, um sistema de software é uma abstração, simplesmente uma representação de algum conhecimento e ideias humanas. Sem o software, o hardware é um conjunto de dispositivos eletrônicos inertes. Entretanto, se você os colocar juntos para formar um sistema, criará uma máquina capaz de realizar cálculos complexos e entregar os resultados desses cálculos para seu ambiente (SOMMERVILLE, 2011).

Com base nestas informações, preencha respectivamente as lacunas abaixo para completar o texto sobre os três tipos principais de sistemas críticos:

No tipo de sistema _____ a falha pode resultar em prejuízos, danos sérios ao meio ambiente e perda de vida humana. E nos _____ a falha resulta em problemas de alguma atividade que possui metas. E nos _____ a falha resulta em custos altos as empresas que usam esse sistema.

Assinale a alternativa que melhor completa as lacunas:

- a) Sistemas Críticos de Segurança, Sistemas Críticos de Proteção, Sistemas Críticos de Negócios.
 - b) Sistemas Sociotécnicos de Software, Sistemas Críticos de Proteção, Sistemas Críticos de Negócios.
 - c) Sistemas Críticos de Segurança, Sistemas Críticos de Missão, Sistemas Críticos de Confiança.
 - d) Sistemas Críticos de Implementação, Sistemas Críticos de Missão, Sistemas Críticos de Negócios.
 - e) Sistemas Críticos de Segurança, Sistemas Críticos de Missão, Sistemas Críticos de Negócios.
5. Conforme Sommerville (2011), ao considerarmos a proteção e confiança do software, é essencial pensarmos de forma holística sobre os sistemas, em vez de apenas considerar o software de forma isolada. As falhas do software por si raramente têm consequências sérias, pois o software é intangível e, mesmo quando danificado, é fácil e barato de ser restaurado. Em sistemas críticos, existem três tipos de falhas que podem ocorrer. Cite e descreva sobre essas três falhas.



REESTRUTURAÇÃO

A reestruturação de software modifica o código-fonte e/ou os dados para torná-lo mais amigável para futuras alterações. Em geral, a reestruturação não modifica a arquitetura geral do programa. Ela tende a concentrar-se nos detalhes de projeto dos módulos individuais e nas estruturas de dados locais definidas nos módulos. Se o trabalho de reestruturação se estende além dos limites de módulo e abrange a arquitetura do software, a reestruturação passa a ser engenharia direta. A reestruturação ocorre quando a arquitetura básica de um aplicativo é sólida, mesmo que as partes técnicas internas necessitem de um retrabalho. Ela ocorre quando partes importantes do software são reparáveis e somente um subconjunto de todos os módulos e dados necessita de uma modificação mais extensa.

Reestruturação de código

A reestruturação de código é feita para obter um projeto que produz a mesma função, mas com mais qualidade do que o programa original. Em geral, as técnicas de reestruturação de código modelam a lógica de programação usando álgebra booleana e aplicam uma série de regras de transformação que resulta na lógica reestruturada. O objetivo é pegar um código “emaranhado” e obter um projeto procedural que esteja em conformidade com a filosofia de programação estruturada.

Reestruturação de dados

Antes de iniciar a reestruturação de dados, deve ser feita uma atividade de engenharia reversa chamada de *análise do código-fonte*. São avaliadas todas as instruções da linguagem de programação que contenham definições de dados, descrições de arquivo, I/O e descrições de interface. A finalidade é extrair itens de dados e objetos, para obter informações sobre fluxo de dados e para entender as estruturas de dados existentes que precisam ser implementadas. Essa atividade às vezes é chamada de *análise de dados*. Uma vez completada a análise de dados, inicia-se o *reprojeto dos dados*.

Engenharia Direta

Um programa com um fluxo de controle, que graficamente equivale a um emaranhado, com módulos de 2 mil instruções, algumas poucas linhas de comentários em 290 mil instruções de código-fonte e nenhuma outra documentação deve ser modificado para acomodar alterações e requisitos de usuário. Temos as seguintes opções:

1. Podemos trabalhar arduamente fazendo modificação após modificação, enfrentando os problemas do projeto *ad hoc* e do código-fonte confuso para implementar as mudanças necessárias.
2. Podemos tentar entender os detalhes internos do programa em um esforço para tornar as modificações mais eficazes.



3. Podemos reprojetar, recodificar e testar as partes do software que requerem modificação, aplicando uma abordagem de engenharia de software a todos os segmentos revisados.
4. Podemos reprojetar completamente, recodificar e testar o programa, usando ferramentas de reengenharia para ajudar a entendermos o projeto atual.

Não há uma única opção “correta”. As circunstâncias podem recomendar a primeira opção mesmo que as outras sejam mais desejáveis. O processo de engenharia direta aplica os princípios, conceitos e métodos de engenharia de software, para recriar um aplicativo. Em muitos casos, a engenharia direta não cria só um equivalente moderno de um programa antigo. Em vez disso, novos requisitos de usuário e tecnologia são integrados ao trabalho de reengenharia. O programa redesenvolvido amplia a capacidade do aplicativo antigo.

Engenharia direta para arquiteturas cliente-servidor

Durante as últimas décadas, muitos aplicativos para mainframes passaram por operações de reengenharia para acomodar arquiteturas cliente-servidor (incluindo WebApps). Essencialmente, recursos de computação centralizados (incluindo software) são distribuídos entre muitas plataformas cliente.

É importante notar que a migração de computação mainframe para cliente-servidor requer reengenharia tanto de negócio quanto de software. A reengenharia para aplicações cliente-servidor começa com uma análise completa do ambiente comercial que abrange o mainframe existente.

Engenharia direta para arquiteturas orientadas a objeto

A reengenharia de software orientada a objeto tornou-se o paradigma de desenvolvimento preferido por muitas organizações. Mas o que dizer sobre os aplicativos desenvolvidos usando--se métodos convencionais? Em alguns casos, a resposta é deixar esses aplicativos “como estão”. Em outros, mais antigos, devem passar por uma reengenharia para que possam ser facilmente integrados em sistemas orientados a objeto de grande porte.

Fonte: Pressman (2016, p. 809).

MATERIAL COMPLEMENTAR



LIVRO

Engenharia de Software – Uma abordagem profissional

Roger S. Pressman

Editora: AMGH

Sinopse: Engenharia de Software chega à sua 8^a edição como o mais abrangente guia sobre essa importante área. Totalmente revisada e reestruturada, esta nova edição foi amplamente atualizada para incluir os novos tópicos da “engenharia do século 21”. Capítulos inéditos abordam a segurança de software e os desafios específicos ao desenvolvimento para aplicativos móveis. Conteúdos novos também foram incluídos em capítulos existentes, e caixas de texto informativas e conteúdos auxiliares foram expandidos, deixando este guia ainda mais prático para uso em sala de aula e em estudos autodidatas.



LIVRO

Engenharia de Software

Ian Sommerville

Editora: Pearson

Sinopse: no intuito de atender às necessidades de alunos e professores dos cursos de ciência da computação, engenharia de computação e sistema de informação, esta nona edição de Engenharia de software teve seu conteúdo reestruturado e totalmente atualizado. A obra conta agora com novos capítulos que focalizam o desenvolvimento ágil de software e os sistemas embarcados, além de trazer novas abordagens sobre engenharia dirigida a modelos, desenvolvimento open source, modelo Swiss Cheese de Reason, arquiteturas de sistemas confiáveis, reuso de COTS e planejamento ágil. Essas novidades somadas à manutenção dos pontos fortes que consagraram a obra, como a clareza de conteúdo e a didática dinâmica, fazem dela um excelente guia para as mais diversas aplicações, independentemente das ferramentas ou processos de software disponíveis ou propostos.



MATERIAL COMPLEMENTAR



NA WEB

Reengenharia

Artigo que fala sobre a reengenharia como uma ferramenta de gestão que tem como objetivo tornar a empresa mais competitiva por meio de medidas que alterem seus processos, ou seja, elimine processos ultrapassados, reinvente novos procedimentos operacionais, o que proporcionará a redução de custos, aumento do grau de satisfação do cliente e aumento da produtividade.

Para conferir, acesse o link: <<http://www.blogdaqualidade.com.br/reengenharia/>>.



NA WEB

O que é engenharia reversa?

Artigo que fala sobre o que a engenharia reversa é e como ela está cada vez mais se tornando uma importante área de pesquisas para o avanço das tecnologias existentes, pois permite que mesmo aquilo que é protegido por leis de propriedade intelectual seja estudado e melhorado pelos concorrentes.

Fique por dentro sobre o assunto acessando o link: <<https://www.tecmundo.com.br/pirataria/2808-o-que-e-engenharia-reversa-.htm>>.



REFERÊNCIAS

PRESSMAN, R.; MAXIM, B. R. **Engenharia de Software** – Uma abordagem profissional. 8. Ed. Porto Alegre: AMGH, 2016.

SOMMERVILLE I. **Engenharia de Software**. 8. ed. São Paulo: Pearson Addison-Wesley, 2007.

SOMMERVILLE, I. **Engenharia de Software**. 9. ed. - São Paulo: Pearson Prentice Hall, 2011.



GABARITO

1. Opção correta é a A.

2. Opção correta é a E.

3. Opção correta é a D.

4. Opção correta é a D.

5. Os três tipos de falhas que podem ocorrer em Sistemas Críticos são:

- Falhas de Hardware: erros de fabricação, final de sua vida útil.
- Falhas de Software: enganos na especificação, projeto ou implementação.
- Falhas Operacionais: falha ao operar o sistema.



REUTILIZAÇÃO DE SOFTWARE

UNIDADE



Objetivos de Aprendizagem

- Entender o que é reutilização de software e como pode ocorrer isso durante o desenvolvimento de software.
- Explorar as vantagens e desvantagens da reutilização de software.
- Estudar a reutilização por meio de: componentes, padrões e frameworks
- Conhecer os principais métodos da engenharia de software baseada em componentes.
- Descobrir as perspectivas futuras de reutilização.

Plano de Estudo

A seguir, apresentam-se os tópicos que você estudará nesta unidade:

- Reutilização de software
- Vantagens e desvantagens da reutilização de software
- Reutilização de componentes, padrões e frameworks
- Engenharia de software baseada em componentes
- A evolução da reutilização de software

INTRODUÇÃO

Caro(a) aluno(a), nesta terceira unidade estudaremos conceitos relacionados à reutilização de software.

O aumento da complexidade dos produtos de software, a crescente demanda por vários tipos de software nas mais diferentes áreas afetando as nossas vidas, a exigência por parte de usuários por produtos de melhor qualidade e com tempo de entrega cada vez menor, fazem com que os desenvolvedores de software busquem por técnicas adequadas para atender às expectativas dos seus clientes.

Ao longo dos tempos, a área de Engenharia de software tem, incessantemente, buscado, desenvolver técnicas e/ou recursos adequados que primam pela entrega de produto de melhor qualidade.

Dentre tais técnicas/recursos temos a reutilização. A ideia de reutilização existe há algum tempo e teve suas origens inspiradas nas ideias de um arquiteto Christoffer Alexander. Além da arquitetura, a reutilização pode ser identificada em outras áreas como a física e matemática.

A área de Ciência da Computação como um todo, também tem buscado explorar a reutilização. No início a reutilização ocorria mais em nível de código. Entretanto com o decorrer dos tempos e amadurecimento, percebeu-se que a reutilização poderia ocorrer em níveis mais altos, o que proporcionou o estudo de componentes, padrões (*design patterns*) e *frameworks*. Todos estes, se bem empregados, oferecem vantagens em termos de tempo de desenvolvimento, custo e, consequentemente, a qualidade.

Atualmente, é praticamente unânime entre os desenvolvedores o entendimento de que é importante o estudo e aplicação de técnicas que proporcionem a reutilização. O assunto é bastante amplo, mas espera-se que ao término do estudo desta unidade vocês possam entender os principais conceitos, as características de cada uma das técnicas e a importância da reutilização.

Então, vamos começar? Boa leitura!



REUTILIZAÇÃO DE SOFTWARE

Os produtos de software são, atualmente, amplamente utilizados nas mais diferentes áreas, e envolvem desde aplicações triviais até sistemas críticos, tais como: sistemas de segurança militar, sistemas de controle aéreo e sistemas de controle financeiro. Logo, a qualidade de um produto de software é uma questão importante, uma vez que estes produtos têm um impacto significativo sobre a sociedade e a vida das pessoas (GIMENES; HUZITA, 2005).

O aumento da complexidade de software e, também, a sua disseminação demandam um desenvolvimento sistemático apoiado por técnicas e mecanismos eficazes que possam ser mensurados e provados, aos seus usuários, que o uso de um determinado produto de software não implicará em riscos.

Os desenvolvedores de software têm buscado por soluções que datam desde a reunião da OTAN em 1968 quando surgiu o termo Engenharia de Software. Dentro as ideias estudadas para apoiar o efetivo desenvolvimento de software tem-se a reutilização de unidades de software resultantes da decomposição de software.

A noção de reutilização é antiga e teve início desde os tempos em que as pessoas começaram a encontrar soluções consistentes para problemas. A motivação estava pautada na ideia de que, uma vez encontrada a solução, esta poderia ser aplicada a novos problemas (GIMENES; HUZITA, 2005).

Ao observarmos as áreas de Matemática, Física e Engenharias, podemos constatar que estas possuem vários exemplos de definições de equações que podem ser utilizadas em várias situações.

A aplicação de uma solução repetida por várias vezes, acaba por torná-la aceita, generalizada e padronizada (PRIETRO-DIAS, 1994). Doug McIlroy já defendia que o uso de biblioteca de programas utilitários para realizar tarefas conhecidas como classificação de itens ou biblioteca de rotinas matemáticas, oferecia muitos benefícios. Este tipo de biblioteca além de economizar esforço de desenvolvimento, permite a produção de programas já testados e amplamente utilizados.

Assim, se essas ideias puderem ser aplicadas a qualquer tipo de sistema, componentes de software identificados como genericamente úteis, poderiam ser armazenados em bibliotecas acessíveis por outros usuários para reutilização, facilitando assim a produção de software.

Ainda, a experiência na indústria de software indicava que muitos requisitos de software não eram novos e teriam sido desenvolvidos, anteriormente, em um contexto um pouco diferente.

A abordagem orientada a objetos que surgiu no final da década de 1980, apresentou soluções que trouxeram muitas esperanças para a reutilização e produção de componentes. Contudo, a ideia de reutilização em larga escala não foi evidenciada na prática.

A área de Engenharia de Software tem buscado estabelecer técnicas, para se definir e projetar soluções que possam ser reutilizadas em diferentes projetos.

DEFINIÇÃO DE REUTILIZAÇÃO

Na literatura podemos encontrar várias definições para reutilização. Vamos conhecer algumas delas:

- Reutilização é o uso de qualquer informação disponível que um desenvolvedor pode necessitar no processo de criação de software (Freeman, 1987).
- Reutilização é a capacidade de um item de software, previamente desenvolvido, ser usado novamente ou usado repetidamente em parte ou todo, com ou sem modificação (Cooper, 1994).
- Reutilização de software é o processo de criar sistemas de software a partir de softwares existentes ao invés de construí-los do zero (Krueger, 1992).

EVOLUÇÃO HISTÓRICA

A Tabela 1 ilustra como os desenvolvedores de software praticaram e evoluíram em termos de reutilização. Pode-se perceber que ao longo dos anos a granularidade foi aumentando.

Tabela 1 - Histórico de reutilização

1960	Reutilização de linhas de código de um programa em outro.
1970	Reutilização de código comum (subrotinas). Reutilização de funções comuns (bibliotecas de funções).
1980	OO: herança, composição e uso de interfaces. Polimorfismo e ligação dinâmica (<i>late binding</i>) qualquer implementação da interface pode ser usada em tempo de execução.
1990	Padrões de software com reutilização de classes e suas relações e não mais apenas em termos de linhas de código. <i>Frameworks</i> : reutilização de análise, projeto, implementação e testes de domínios de aplicações. Componentes: reutilização de código executável, configurável e adaptável.
2004	Serviço: reutilização de unidade autônoma de execução (função de negócio).
2009	Ideia de ecossistemas de software.

Fonte: adaptado de Martins (2005).

VANTAGENS E DESVANTAGENS DA REUTILIZAÇÃO DE SOFTWARE

Segundo Sommerville (2011), uma vantagem óbvia da reutilização de software é a redução dos custos globais de desenvolvimento. Menos itens precisam ser concebidos, especificados, implementados e validados. Além da vantagem da redução de custo, podem-se destacar outros benefícios com a reutilização, conforme consta na Tabela 2.

Tabela 2 – Benefícios da Reutilização

BENEFÍCIO	EXPLICAÇÃO
Aumento da confiabilidade	Reutilizam-se soluções já prontas
Redução de riscos do processo	A reutilização de componentes prontos reduz as incertezas quanto aos custos de desenvolvimento.
Uso eficaz de especialistas	Em vez de repetir o mesmo trabalho, especialistas em aplicações podem desenvolver produtos de software que encapsulem seu conhecimento.
Conformidade com padrões	Alguns padrões como os de interface de usuário, podem ser implementados como um conjunto de componentes reutilizáveis. Por exemplo, se os menus em uma interface de usuário forem implementados usando componentes reutilizáveis, todas as aplicações apresentarão os mesmos formatos de menu para os usuários.
Desenvolvimento acelerado	A reutilização pode acelerar a produção de um sistema graças à redução do tempo de desenvolvimento e validação.

Fonte: Sommerville (2011, p. 297).

Da mesma forma, ainda conforme Sommerville (2011), a reutilização pode apresentar algumas dificuldades, conforme ilustradas na Tabela 3.

Tabela 3 – Dificuldades da Reutilização

PROBLEMAS	EXPLICAÇÃO
Aumento nos custos de produção	A ausência/não disponibilidade do código fonte dos artefatos reutilizáveis pode aumentar os custos de manutenção uma vez que, os elementos reutilizados podem se tornar cada vez mais incompatíveis com as alterações do sistema.
Síndrome do “não inventado aqui”	Alguns engenheiros de software preferem reescrever o código, pois acreditam poder melhorá-lo. Isto tem a ver, parcialmente, com o aumentar a confiança e, para alguns ser mais desafiador do que reutilizar o que foi desenvolvido por outras pessoas.
Criação, manutenção e uso de uma biblioteca	Preencher uma biblioteca de artefatos reutilizáveis e garantir que desenvolvedores possam utilizar essa biblioteca podem ser ações caras. É preciso criar esta cultura.

PROBLEMAS	EXPLICAÇÃO
Busca, compreensão e adaptação de componentes reutilizáveis	Componentes de software precisam ser descobertos em uma biblioteca, compreendidos e, às vezes, adaptados para trabalhar em um novo ambiente. Os engenheiros precisam estar confiantes de que encontrarão, em uma biblioteca, um componente antes de incluírem a pesquisa de componentes como parte de seu processo de desenvolvimento.

Fonte: Sommerville (2011, p. 297).

Portanto, para uma efetiva reutilização, é importante que se atenda a alguns requisitos:

1. Existência de uma biblioteca (ou repositório de componentes)
ex.: *component source, jars*;
2. Garantia de que o componente se comportará conforme foi especificado e que serão confiáveis;
3. Existência de documentação que ajude a compreendê-los e adotá-los (MARTINS, 2005).

REUTILIZAÇÃO DE COMPONENTES, PADRÕES E FRAMEWORKS

Uma vez que a reutilização pode ocorrer em diferentes níveis de abstração, cabe apresentar a definição de componentes, padrões e *frameworks* e como pode ser visto a reutilização em cada um destes.

REUTILIZAÇÃO DE COMPONENTES

Antes de falarmos em reutilização de componentes, é importante que você, caro(a) aluno(a), entenda o que é componente. Então vejamos a definição de Gimenes e Huzita (2005): “um componente é uma unidade de software independente, que encapsula, dentro de si, seu projeto e implementação e oferece serviços, por meio de interfaces bem definidas para o meio externo”.

Interfaces de componentes

Um componente deve oferecer uma especificação clara de seus serviços, o que é feito por meio da interface. Cada interface consiste em serviços especificados, mediante uma ou mais operações, sendo cada uma delas separadamente identificadas e especificadas de acordo com seus parâmetros de entrada, saída e tipos estabelecidos. Essas definições constituem o que é conhecido como assinatura da interface.

As interfaces podem ser de dois tipos: interfaces fornecidas (*provided interfaces*) e as interfaces requeridas (*required interfaces*). As interfaces fornecidas definem os serviços oferecidos pelo componente por meio de operações. Já as interfaces requeridas se referem aos serviços que o componente necessita de outros componentes. Portanto, os componentes se conectam por meio da interface requerida de um com a interface fornecida de outro.

Notação gráfica

A ideia de componentes usando notação UML (*Unified Modeling Language*) para pacotes e interfaces já tem algum tempo (Rumbaugh, Jacobson e Booch, 1998) quando as interfaces eram representadas pelos relacionamentos de dependência. A evolução da notação UML inseriu a representação de interfaces requeridas e providas conforme ilustra a figura 1. Neste exemplo, tem-se o componente Pedido que provê as interfaces Alocação de Itens e Rastreamento. Por outro lado, Pessoa, Fatura, Item Disponível são interfaces requeridas.

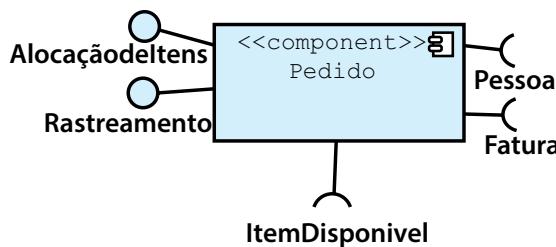


Figura 1 - Componente Pedido e suas interfaces
Fonte: a autora.

COTS (*commercial off the shelf*)

Além de projeto e especificação de componentes para aplicações organizacionais, existe também um interesse por componentes comerciais ou componentes COTS (*commercial off the shelf*). Isto envolve pesquisas em formas de produzir, empacotar, regulamentar e selecionar componentes para produzir sistemas a partir de componentes comerciais existentes.

Em muitos sistemas construídos o principal desafio é utilizar o máximo de componentes comerciais de software (WALLNAU; HISSAM; SEACOD, 2001). Exemplos de componentes comerciais incluem: navegadores (*Web Browsers*), servidores HTTP, ORB, *Middleware* orientado a mensagens dentre outros.

Pode-se ainda citar alguns benefícios/problemas do uso de componentes, além daqueles já decorrentes da própria reutilização conforme listados nas tabelas 1 e 2.

Benefícios

- Gerenciamento de complexidade: lidando com um número reduzido de componentes de cada vez, torna possível gerenciar a complexidade do sistema, reduzindo portanto os riscos de desenvolvimento.
- Desenvolvimento paralelo: a decomposição do sistema em componentes permite a definição de componentes independentes, que podem ser subcontratados pelo menos em parte.
- Facilidade de manutenção e atualização: o fato do sistema ser construído a partir de componentes facilita a localização de modificações e atualizações.

Problemas:

- Seleção do componente certo: encontrar o componente certo disponível não é uma tarefa fácil.
- Cultura dos engenheiros de software: alguns são relutantes em confiar em código desenvolvido por terceiros e querem ter o controle do desenvolvimento de software (Gimenes; Huzita, 2005).

Segundo Sommerville (2011, p. 312), o reuso de produtos COTS está preocupado com o reuso de sistemas de prateleira em grande escala. Eles fornecem uma grande funcionalidade e reduzir, radicalmente, os custos e o tempo de desenvolvimento. Os sistemas podem ser desenvolvidos por meio da configuração de um único produto genérico COTS ou integrando dois ou mais produtos COTS.

Os conhecidos sistemas ERP (*Enterprise Resource Planning*) são exemplos de reuso COTS em grande escala. De um modo geral, cria-se uma instância de um sistema ERP por meio da configuração de um sistema genérico com informações sobre regras e processo de negócios do cliente.

Assim, possíveis problemas de reuso com base em COTS incluem a falta de controle sobre a funcionalidade e o desempenho, a falta de controle sobre a evolução do sistema, a necessidade de suporte dos fornecedores externos e às dificuldades em assegurar que os sistemas possam interoperar.

A materialização de componentes prontos pode se dar de duas formas (Martins, 2005):

- Reutilização de componente já utilizado pela organização: Geralmente os componentes associados ao domínio de negócio são os mais propícios a serem reutilizados
- Aquisição de componentes a partir de catálogos de terceiros: Compra ou uso de software livre (*open source* ou *freeware*)

No entanto, pode ocorrer que componentes reutilizados nem sempre atendem exatamente aos requisitos. Assim, pode ocorrer a necessidade de adaptar os requisitos e, também, a arquitetura para permitir a integração de componentes prontos.

No entanto, é importante que se observe com cuidado quando se refere a custo e tempo de desenvolvimento. Da mesma forma que a reutilização pode ajudar (**desenvolvimento com reuso**) também requer esforço adicional, esse esforço se deve à possível reutilização futura dos componentes (**desenvolvimento para reuso**) e à necessidade de serem flexíveis, estáveis e corretos. Tal esforço é consideravelmente superior ao necessário para qualquer outro software desenvolvido para uma aplicação específica.

Documentação de componentes

A documentação de um componente deve incluir pelo menos:

- A especificação do componente. Pode ser: (i) uma especificação de uso, que inclui a especificação da interface que consiste em uma lista de operações fornecidas com suas assinaturas mais um modelo de estados; (ii) especificação de implementação que inclui as interfaces requeridas que podem ser definidas por diagramas de colaboração, e é vista como a parte da especificação do componente que não precisa ser conhecida pelo usuário.
- O relatório de validação, consistindo em uma série de testes que validam os componentes nos ambientes para os quais eles foram projetados.
- As propriedades não funcionais pertinentes ao componente, como segurança, desempenho e confiabilidade.

Além da especificação de uso e de implementação, deve-se, também, especificar os conectores. Constituem-se exemplos de conectores: chamada explícita de procedimentos (síncrona e assíncrona), propagação de eventos, colocação de objetos em uma estrutura de dados de saída a partir da qual podem ser consumidos por outros componentes, *workflow* em que os objetos são transferidos entre componentes e códigos móveis em que objetos completos com o código que define o seu comportamento são transmitidos entre componentes (D'SOUZA e WILLS, 1998).

Arquitetura de software

A definição de componentes em desenvolvimento de software desperta o interesse em identificar como estes podem ser representados, ligados e usados. Uma definição e, também, uma área que tem despertado interesse de estudiosos refere-se a arquitetura de software. Assim, na literatura podem ser encontradas várias definições para arquitetura de software como as que seguem:

- É um conjunto de componentes computacionais e os relacionamentos entre esses componentes (SHAW; GARLAN, 1996).

- É a estrutura ou estruturas do sistema que compreende elementos de software, as propriedades externamente visíveis destes elementos e os relacionamentos entre eles (Bass, et al, 2003).
- Uma arquitetura de software pode ser vista como a estrutura geral de um sistema de software, seus componentes e como eles se conectam para garantir funcionalidade e qualidade ao software (Barroca, Hall e Hall, 2000).

Existem estilos clássicos de arquitetura de software que definem certos tipos de componentes e a forma como eles interagem. Um estilo de arquitetura descreve uma classe de arquiteturas de software. O estilo restringe os componentes usados e a forma como esses componentes podem ser conectados. Por exemplo, em arquiteturas do tipo tubos e filtros, os componentes são os filtros que transformam sequências de entrada em sequências de saída, as interfaces são os tubos que transmitem a saída de um filtro para outro, os conectores definem como os dados fluem entre os tubos.

Os estilos arquiteturais refletem boas práticas de construção de software e foram desenvolvidos para aumentar a qualidade dos sistemas de software. O uso de estilos estabelecidos é uma manifestação de reutilização de experiências.

A arquitetura de software é importante uma vez que: (i) ajuda a entender a visão global, (ii) ajuda a organizar o esforço de desenvolvimento, (iii) facilita as possibilidades de reuso, (iv) facilita a evolução do sistema, (v) melhora a comunicação da equipe e, (vi) é um importante meio de comunicação com *stakeholders*.

Para a proposição de uma arquitetura, pode-se seguir alguns passos: (i) identifique os principais componentes da aplicação e qual o papel deles, (ii) identifique as interfaces ou serviços que cada componente apoia, (iii) identifique as responsabilidades do componente, afirmando quais ações são garantidas uma vez que ele receba uma requisição, (iv) identifique dependências entre componentes e, (v) identifique partições na arquitetura que sejam candidatas para distribuição entre os servidores de uma rede

Como se pode observar, existe uma estreita relação entre componentes e arquitetura de software. Isto justifica, de uma certa forma, o crescente interesse por estudos sobre arquitetura de software, arquitetura de referência (Nakagawa, 2006).

Comentários interessantes sobre componentes

- Os componentes que utilizam outro componente devem fazê-lo com base apenas nas interfaces definidas e serviços especificados, não sendo feita suposição alguma sobre a sua implementação.
- Componentes podem ser interconectados para formar componentes maiores, o que faz dos componentes estruturas recursivas.
- O desenvolvimento para e com reutilização é uma tarefa complexa que tem consequências tanto técnicas quanto gerenciais e organizacionais.
- Além da reutilização, a produção de componentes torna-se relevante uma vez que aumentam as demandas por liberação de produtos para o mercado (*time to market*) além da necessidade de lidar com modificações de forma rápida e efetiva.
- Um repositório de componentes pode ser entendido como sendo uma base preparada para o armazenamento e recuperação de componentes. Os repositórios podem ser: (i) repositórios locais que armazenam componentes de propósito geral em um repositório único; (ii) repositórios específicos a um domínio: armazenam componentes específicos com escopo bem definido; (iii) repositórios de referência: auxiliam na busca por componentes em outros repositórios, funcionando como uma espécie de páginas amarelas (Sametinger, 1997).
- Em se tratando de repositórios de componentes um ponto importante a considerar é como se dão a busca e a seleção dos componentes armazenados. Essa busca e seleção estão intimamente relacionadas com a qualidade dos mecanismos de classificação dos componentes.
- Um outro ponto importante é a análise da qualidade de componentes. Assim, são atributos importantes: segurança, desempenho, confiabilidade, qualidade da arquitetura de software.
- Outra preocupação é a adoção efetiva do DBC (Desenvolvimento Baseado em Componentes), evolução e adaptação dos componentes a fim de que possam ser reutilizados em diferentes contextos.

REUTILIZAÇÃO COM PADRÕES DE PROJETO

Caro(a) aluno(a), aqui nesta subseção iremos também inicialmente definir padrões de projeto para então entendermos a reutilização com padrões.

Definição:

Um padrão descreve um problema que ocorre repetidamente no nosso ambiente, descrevendo a essência de uma solução para este problema, de forma que pode-se usar esta solução milhares de vezes, sem fazê-lo da mesma forma duas vezes (ALEXANDER et al., 1977).

Um padrão de software nomeia, motiva e explica uma solução genérica a um problema recorrente que surge em uma situação específica. Ele descreve o problema, a solução, quando é aplicável e quais as consequências de seu uso (GAMMA et al., 2002, p. 395).

Características de patterns

- Descrevem e justificam **soluções** para problemas concretos e bem definidos.
- Documentam a experiência **existente e comprovada**.
- Fornecem um **vocabulário** comum aos desenvolvedores de software.
- Descrevem **relações** entre conceitos, estruturas e mecanismos existentes nos sistemas.
- Podem ser utilizados em **conjunto** com outros padrões. (MARTINS, 2005)

Os padrões, no contexto de desenvolvimento de software, são denominados *design patterns*. Estes têm como objetivo descrever soluções para problemas recorrentes no desenvolvimento de software. Com *design patterns* a busca é pelo reuso de solução e não apenas de código.

Os *design patterns* têm foco nas fases de projeto e implementação, mas, também, podem ser encontrados padrões de análise que são artefatos de alto nível.

Os seguintes fatores têm contribuído para motivar o uso de *design patterns* em projetos:

- Com a reutilização possibilitar redução de tempo e custo de desenvolvimento e, consequentemente, aumentar a produtividade.
- Evitar falhas uma vez que os componentes foram previamente testados.
- Interoperabilidade: componentes de diferentes origens podem compartilhar e trocar informações.
- Aumentar a possibilidade de reutilização de boas soluções para problemas freqüentes, garantindo, com isso, melhor qualidade para o software.
- Ser possível reutilizar soluções para problemas surgidos em trabalho anterior.
- Os padrões surgem com a experiência prática.
- Experiência de especialistas pode ser compartilhada com novatos.

Ainda, o uso de *design patterns* traz vantagens tais como: aprender com a experiência de outros; utilizar vocabulário comum entre os desenvolvedores; facilitar a documentação e aprendizagem e, aumentar a qualidade do produto gerado.

Todo *design pattern* possui, basicamente, os seguintes elementos como uma forma de documentá-lo e, assim, facilitar o uso pelos desenvolvedores:

- **Nome** [descreve a essência do padrão, deve ser curto e expressivo].
- **Problema** [descreve a intenção e objetivo].
- **Contexto** [define a configuração inicial, antes do padrão].
- **Forças** [destaca a motivação pelo seu uso].
- **Solução:**
 - Estrutura [apresenta a organização estática do padrão].
 - Participantes [explicita classes e objetos].
 - Dinâmica [define o comportamento].
 - Implementação.
 - Variantes [apresenta especializações da solução].

- **Exemplos**
- **Contexto resultante**
- **Padrões relacionados**
- **Usos conhecidos**

Catálogo de Gof (Gang of four – Gamma (2002))

Com o passar dos tempos, vários padrões estavam sendo propostos pelos envolvidos em desenvolvimento de software. Com isto, veio a necessidade de reunir e catalogar as soluções de projeto que estavam surgindo.

A catalogação constitui-se em uma das primeiras iniciativas e foi estruturada de acordo com a *finalidade* e o *escopo* de cada um dos padrões. Por finalidade buscou-se refletir o que a solução faz. Tomou-se o cuidado de classificá-los como sendo padrões de criação, estruturais e comportamentais. Já quanto ao escopo, a ideia foi identificar as entidades sobre as quais o padrão se aplica: se classes ou objetos (GAMMA et al., 2002). Tal catalogação é apresentada na Tabela 4.

Tabela 4 - Catalogação de padrão

	CRIAÇÃO	ESTRUTURAL	COMPORTAMENTAL
CLASSE	Factory Method	Adapter	Interpreter Template method
OBJETO	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Fonte: Gamma (et al, 2002).

Como usar um padrão (Deschamps, 2005):

Uma vez que você tem acesso a um conjunto de diversos padrões, vem a pergunta: como faço para usar um padrão? A seguir, são apresentadas algumas diretrizes que podem te ajudar a nortear o uso.

- Leia o padrão por inteiro, uma vez, para obter uma visão geral.
- Estude as seções de descrição do problema e do padrão.
- Olhe exemplos de código do padrão.
- Escolha nomes para os participantes do padrão que tenham sentido no contexto da sua aplicação.
- Defina as classes.
- Defina nomes específicos da aplicação para as operações no padrão.
- Implemente as operações para apoiar as responsabilidades e colaborações presentes.

Conforme Martins (2005), o uso de padrões pode melhorar a compreensão do código e, consequentemente, melhora a modularidade, separação de conceitos e simplicidade de descrição.

Assim, por exemplo: é mais fácil dizer: “utiliza-se uma instância do padrão *Visitor*” do que “este é um código que varre a estrutura e realiza chamadas a alguns métodos em uma ordem particular e de uma determinada forma”.

Conforme consta em (Gimenes e Huzita, 2005, p. 218) além da classificação acima apresentada (Gof), a literatura traz também mais uma classificação que foi proposta por (Buschman, 1996). Constitui-se de uma lista de padrões, conhecida como POSA (*Pattern oriented software architecture*). Os padrões são agrupados conforme as seguintes categorias:

- Padrões arquiteturais: expressam um esquema de organização estrutural e fundamental para um sistema. São conjuntos de subsistemas pré-definidos, especificando seus relacionamentos e as regras para a sua organização.

- Padrões de projeto: refinam os subsistemas ou componentes de um sistema, descrevendo uma estrutura que constitui uma solução para um problema de projeto.
- Idioma: são padrões específicos para uma linguagem de programação. Descrevem a forma de implementar um aspecto particular de um componente usando características de uma linguagem específica.

REUTILIZAÇÃO COM FRAMEWORKS

Prezado(a) aluno(a), assim como definimos inicialmente componentes e padrões de projeto antes de falarmos em termos de reutilização, procederemos da mesma forma com frameworks. Segue então uma definição:

[...] um *framework* é *um conjunto cooperativo de classes que tornam um projeto reutilizável para uma classe específica de software*. Um *framework* fornece uma orientação arquitetural através da divisão do projeto em classes abstratas e definindo as suas responsabilidades e colaborações. Um desenvolvedor irá customizar um *framework* para uma aplicação particular através de subclasses e compor instâncias de classes do *framework* (GAMMA, et al, 2002, p. 395).

SAIBA MAIS



Historicamente o conceito de padrões não foi concebido por profissionais da área de Ciência da Computação.

O arquiteto, Christopher Alexander (1970) escreveu dois livros sobre padrões de projeto para a arquitetura civil: "*The timeless way of building*" (1977) e "*A pattern language: Towns, Buildings, Construction*" (1979)". Alexander encontrou temas recorrentes na arquitetura (planejamento urbano e arquitetura de prédios) e os capturou em descrições e instruções que ele chamou de padrões.

Fonte: a autora.

Para Martins (2005):

- *Framework*: projeto de um subsistema (ou arquitetura de software semi-definida) constituído de um conjunto de componentes individuais e das interconexões entre eles.
- *Frameworks* criam uma infra-estrutura pré-fabricada para o desenvolvimento de aplicações de um domínio específico.
- Uma aplicação pode ser construída a partir da integração de vários *frameworks*.

Ainda, segundo Martins (2005), em um framework podem ser identificados:

- **Hot spots**: constituem-se em partes do framework que são projetados para serem genéricos e, podem ser adaptados às necessidades da aplicação.
- **Frozen spots**: Definem a arquitetura geral da aplicação, seus componentes e os relacionamentos entre eles. Eles permanecem fixos em todas as instanciações do framework.

Conforme o modo de estender um *framework* tem-se os seguintes tipos:

- **Framework caixa branca**: usuários desenvolvem implementações para as classes abstratas fornecidas, que funcionam como **pontos adaptáveis**, bem como o código específico para a aplicação.
- **Framework caixa preta**: contém todas as possíveis alternativas para todos os seus pontos adaptáveis. O usuário escolhe uma das alternativas disponíveis para criar sua aplicação (MARTINS, 2005).

Fayad (1999) classifica *frameworks* de aplicação de acordo com seu contexto da seguinte forma:

- *Framework* de infraestrutura de sistema: são *frameworks* utilizados como sistemas operacionais.
- *Frameworks* de integração e comunicação (middleware): são *frameworks* utilizados para integrar aplicações e componentes distribuídos, como por exemplo os ORBs (*Object Request Broker*).
- *Framework* de aplicações: são *frameworks* que tem no domínio de aplicações.

SAIBA MAIS



O modelo *Model-View-Control* (MVC) é um modelo de interação dos mais conhecidos, mais antiga e talvez a mais utilizada, tendo sido originada no ambiente *Smalltalk* (apud, Lewis, 1995).

Fonte: Lewis (1995 apud GIMENES; HUZITA 2005, p. 18).

A estrutura MVC está ilustrada na figura a seguir:

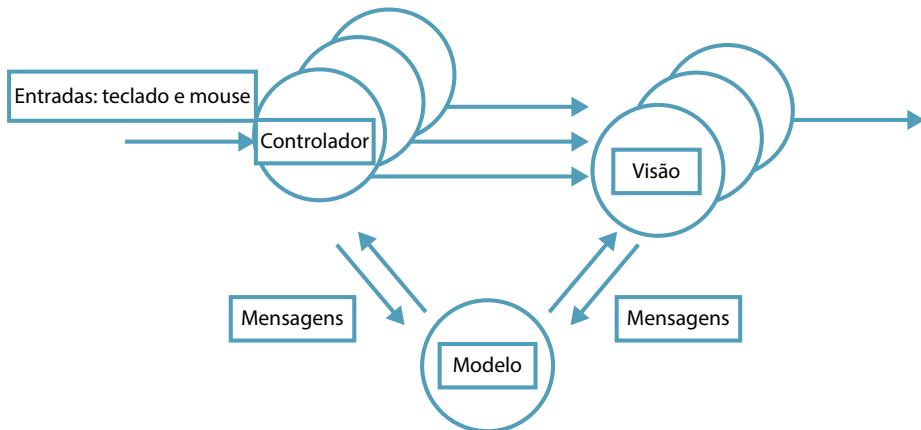


Figura 2 - Modelo Model-View-Control

Fonte: Gimenes e Huzita (2005).

A partir da estrutura proposta no modelo MVC, pode-se construir um *framework* de interface com o usuário (GUI). Esse *framework* tem um conjunto de classes sendo os principais: o Modelo, a Visão e o Controlador. A classe Modelo contém os dados específicos do domínio que serão representados e manipulados a partir das interfaces com o usuário. A classe Visão, por sua vez, é responsável por mostrar os dados ao usuário. A classe Controlador aceita entradas assíncronas, do mouse ou teclado e passa as mensagens apropriadas para as classes Visão e Modelo de modo a permitir a edição dos dados da aplicação.

A proposta de reutilização aqui está no fato do projetista adotar a estrutura de organização do sistema em 3 partes – o Modelo, a Visão e o Controlador – ao invés de iniciar um novo projeto e propor, por exemplo, uma estrutura que misture as classes de interface com o usuário e com as classes do domínio da aplicação. O projetista encontrará, por meio do framework, um conjunto de classes pré-concebidas, a partir das quais pode iniciar a implementação de uma aplicação.



REFLITA

Um *framework* pode conter diversos *design patterns*, mas o contrário nunca é verdadeiro.

(Gudwin)

ENGENHARIA DE SOFTWARE BASEADA EM COMPONENTES

O interesse por Desenvolvimento Baseado em Componentes (DBC) advém da maturidade das tecnologias que permitem a construção de componentes e a combinação destas para a construção de sistemas, bem como o atual contexto organizacional e de negócio que apresenta mudanças à forma como as aplicações são desenvolvidas, utilizadas e mantidas.

O desenvolvimento de software baseado em reutilização é visto como um ciclo contínuo de aprendizagem influenciado por atividades de Engenharia de domínio (ED) (Arango, 1988).

De acordo com Blois (2006), o objetivo da ED é possibilitar que características comuns e variáveis possam ser identificadas e modeladas com base em um processo previamente definido. Os artefatos gerados pela ED (e.g. modelos de domínio, arquiteturas de componentes, casos de uso, dentre outros) podem ser instanciados para uma aplicação específica do domínio. A esta instanciação se dá o nome de Engenharia de Aplicação (EA). A ED provê um conjunto de artefatos “para” reutilização, enquanto que a EA constrói aplicações “com” base na reutilização de artefatos providos pela ED.

Um conceito importante que surge é o que se refere a Engenharia de Software Baseada em Componentes (ESBC) ou *component based software engineering* (CBSE do inglês). Assim, de acordo com (Sommerville, 2011 p. 315) ESBC é uma abordagem para desenvolvimento de sistemas de software com base no reuso de componentes.

Na literatura, existem propostas de metodologias de Desenvolvimento Baseado em Componentes (DBC), dentre as quais serão descritas, a seguir, três delas, conforme consta em (Gimenes e Huzita, 2005):

UML COMPONENTS (CHEESMAN E DANIELS, 2001)

Foi proposto como uma modificação do RUP (*Rational Unified Process*) para atender aspectos específicos de Desenvolvimento Baseado em Componentes (DBC). Nessa modificação, foram incluídas ou modificadas atividades como a especificação, o fornecimento e a montagem (as quais substituem, respectivamente, os processos de análise, projeto e implementação originais do RUP).

As adaptações feitas aproveitam as técnicas fornecidas pela própria UML (*Unified Modeling Language*) como a utilização de estereótipos e OCL (*Object Constraint Language*).

Ao final é gerado um conjunto de modelos UML modificados para incluir características DBC.

UML Components é constituído dos seguintes workflows:

a) Requirements workflow (workflow requisitos)

Com base no conhecimento do domínio, requisitos do negócio e a definição da delimitação do software, são elaborados os seguintes modelos: (i) Modelo conceitual das informações que existem no domínio do problema (*Business Concept Model Diagram*), (ii) Diagrama com os tipos do negócio (e seus relacionamentos) que precisam ser mantidos pelo software (*Business Type Diagram*) e, (iii) Diagrama de casos de uso do software.

b) Specification workflow (workflow especificação)

Tem como entrada: um modelo de caso de uso e um modelo conceitual de negócios vindos do workflow de requisitos. É gerado um conjunto de especificação de componentes e uma arquitetura de componentes. A especificação de componentes inclui a especificação da interface que eles suportam ou dependem. A arquitetura de componentes mostra como os componentes interagem uns com os outros.

Portanto, são gerados os seguintes modelos: (i) Diagrama com uma definição precisa sobre as ações de uma interface, possuindo um modelo de informação, a especificação das operações e as invariantes (*Interface Specification Diagram*), (ii) Diagrama que descreve as interfaces fornecidas e exigidas de uma especificação de componente (*Component Specification Diagram*), (iii) Diagrama da arquitetura de componentes (*Component Architecture Diagram*),

(iv) Diagrama que mostra o conhecimento de uma interface sobre os elementos de negócio (*Interface Responsibility Diagram*), (v) Diagrama de interação entre uma especificação de componente e suas interfaces em tempo de execução (*Component Interaction Diagram*).

c) *Provisioning workflow* (workflow provisão/fornecimento)

Determina quais componentes construir ou comprar. Desta forma assegura que os componentes necessários estarão disponíveis quer seja construindo-os do zero ou comprando-os de terceiros.

Componentes existentes podem ser reusados, integrados ou modificados.

De qualquer forma, são realizados testes dos componentes antes da montagem.

d) *Assembly workflow* (workflow de montagem)

Aqui deve ocorrer a correta composição dos componentes. Para tanto, colocam-se todos os componentes juntos com o software existente e uma adequada interface com usuário para formar uma aplicação executável que vai ao encontro das necessidades de um negócio.

e) *Test workflow* (workflow Teste)

f) *Deployment workflow* (workflow entrega)

Os demais *workflows* podem gerar modelos conforme disponibilizados pela UML e *Rational Unified Process* (RUP).

CATALYSIS (D'SOUZA, E WILLS, 1998)

Catalysis é um método DBC que apresenta conceitos importantes para o desenvolvimento de software baseado em componentes. Os princípios fundamentais (D'SOUZA e WILLS, 1998; GIMENES e HUZITA, 2005) são:

- Construção de um modelo de domínio do sistema, com base em tipos, objetos e ações que enfatiza a independência entre o domínio, a possível solução de software e a sua implementação.

- Forte ênfase nos conceitos de abstração e refinamento para representar os relacionamentos essenciais entre os artefatos do sistema obtidos durante o processo de desenvolvimento. A ênfase no refinamento dos artefatos cria uma série de fatorações, extensões e transformações que visam o rastreamento dos requisitos até o código.
- Ênfase na especificação de pré-condições, pós-condições e invariantes apoiando-se em OCL (Warmer e Kleppe, 1998).
- Procedimentos e diagramas que apoiam o particionamento do sistema, o projeto e a conexão dos componentes.
- Forte articulação do processo de desenvolvimento com os conceitos de arquitetura de software, padrões e frameworks.
- Uso de ciclo de vida rápido, iterativo e incremental.

Na sequência, vamos identificar os modelos elaborados quando se adota *Catalysis*, mapeando-os aos estágios de um processo de desenvolvimento.

- Especificação de requisitos: tem por objetivo entender o contexto do sistema, na arquitetura e os requisitos funcionais. Gera-se o Modelo do domínio. Entende-se o contexto do sistema.
- Especificação de sistema: tem por objetivo descrever o comportamento externo do sistema por meio do modelo de domínio do problema. Geram-se cenários; modelo de tipos e especificação de operações.
- Projeto da arquitetura: tem por objetivo separar os componentes arquiteturais técnicos dos de aplicação e os seus conectores para se alcançar os objetivos de projeto. A arquitetura de aplicação se preocupa em decompor a própria aplicação em um conjunto de componentes que colaboram entre si. É na verdade um projeto lógico de alto nível. Seus modelos são totalmente independentes da tecnologia de suporte. Inclui-se aqui também a adoção de um estilo arquitetural. Já a arquitetura técnica é totalmente independente do domínio da aplicação e trata dos meios de comunicação que são utilizados para possibilitar a comunicação entre os componentes da arquitetura da aplicação. Detalhes de hardware, protocolos de comunicação e padrões de colaboração são tratados aqui. Componentes podem ser acoplados de várias maneiras dependendo do projeto. Soluções como CORBA, Java Beans, protocolos de interface com banco de dados podem ser usadas.

- Projeto interno de componentes: tem por objetivo projetar interfaces e classes para cada componente, construir e testar. Gera-se a especificação de classes e interface.

O *Catalysis* identifica os pacotes como a unidade de decomposição de mais alto nível. Esses pacotes podem ser obtidos pela análise e refinamento do modelo de negócios. Portanto, tem um enfoque forte em particionamento do sistema em componentes, o que é crucial para DBC.

O método propõe algumas técnicas de particionamento desses pacotes e do sistema como um todo da seguinte forma:

- Divisão vertical: visa refletir o fato de que usuários diferentes têm visões diferentes de um sistema ou negócio.
- Divisão horizontal: visa organizar os pacotes em camadas horizontais desde as ações de mais alto nível até as de infra-estrutura.
- Domínios diferentes: visam separar domínios de funcionalidades diferentes, como interface com o usuário, persistência ou subdomínios do próprio problema.

KOBRA (ATKINSON ET AL., 2000)

Foi criada com o objetivo de tornar os componentes de um sistema de software o foco do processo de desenvolvimento. Adota uma estratégia de linha de produtos para a criação, manutenção e implantação de componentes. Utiliza uma visão de componentes que são descritos em uma representação baseada em UML, que expressa suas características e relacionamentos. Isto faz com que as atividades de análise e projeto se tornem orientadas a componentes e, também, permite que a estrutura e o comportamento dos sistemas possam ser descritos independente de tecnologia.

Por tratar de linha de produtos, a abordagem Kobra utiliza o conceito de *framework* genérico e reutilizável que é utilizado para instanciar aplicações. A utilização dessa abordagem de *frameworks* genéricos é que faz com que a abordagem Kobra utilize conceitos adotados em abordagens ED.

Cada componente nesta abordagem é dividido em duas partes: a *especificação* que descreve as características externamente visíveis do componente (isto é, os requisitos) e a *realização* que descreve como o componente satisfaz essas características em termos de interações com outros componentes, descrevendo inclusive o projeto interno.

A EVOLUÇÃO DA REUTILIZAÇÃO DE SOFTWARE

No decorrer dos últimos anos percebeu-se um esforço, por parte de estudiosos da área, para tornar a reutilização uma realidade cada vez mais presente na vida de desenvolvedores. Assim, tivemos a vertente da Linha de Produtos de Software e, mais recentemente, Ecossistemas de Software.

LINHA DE PRODUTOS DE SOFTWARE (LPS) OU FAMÍLIA DE APLICAÇÕES

Uma linha de produto de software envolve um conjunto de aplicações similares dentro de um domínio, que pode ser desenvolvido a partir de uma arquitetura genérica comum, a arquitetura da linha de produto, e um conjunto de componentes que povoam a arquitetura (GIMENES;TRAVASSOS, 2002).

Esta abordagem tem por objetivo identificar os aspectos comuns e as diferenças entre os artefatos de software ao longo do processo de desenvolvimento da linha de produto, de modo a explicitar os pontos de decisão em que a adaptação dos componentes para geração de produtos específicos pode ser realizada.

Para tal, durante o processo de desenvolvimento deve-se identificar os pontos de variabilidade que são pontos em que as características dos produtos podem se diferenciar. Esses pontos de variabilidade aparecem inicialmente na definição dos requisitos da linha de produto e devem ser representados ao longo de todo

o processo de desenvolvimento. Dessa forma, pode-se decidir por uma característica ou outra para um produto específico tanto em nível de projeto quanto em nível de implementação.

Frameworks e patterns também são importantes recursos para o enfoque de linha de produto. Para se conceber uma linha de produto é necessário partir de conceitos que levem à generalização de aplicações e que permitam instâncias para aplicações específicas. Os frameworks fornecem esse tipo de recurso. Da mesma forma, os *patterns* permitem que a experiência de desenvolvedores de software seja documentada e reutilizada, registrando-se soluções de projeto para um determinado problema ou contexto particular. Portanto, o enfoque de linha de produtos encontra nos padrões indicações de como construir tanto a arquitetura quanto os componentes desta.

As três atividades essenciais da construção de uma linha de produto são (CLEMENTS; NORTHORP, 2002, apud MARTINS (2005)):

- Desenvolvimento de ativos centrais: os ativos centrais constituem-se nas partes a serem reutilizadas entre os produtos da família
- Desenvolvimento de produtos: Construção de produtos utilizando os ativos centrais
- Gerência: Atividades que gerenciam a implantação e manutenção de LPS

ECOSISTEMA DE SOFTWARE

Um sistema de software representa uma combinação de software, hardware e “*peopleware*”, constituída sobre um ambiente comum, isto é, uma plataforma.

Uma vez que uma empresa leva seus produtos de software além de seus limites organizacionais, passando a disponibilizar a sua plataforma e a interagir com atores externos a sua organização, forma-se um ECOS (BOSCH, 2009).

Assim, um conjunto de elementos configura os ECOSs: (i) os atores envolvidos dentro e fora da organização, (ii) o produto de software principal, (iii) a plataforma de apoio ao software e (iv) os ativos de ECOS. Estes elementos são tratados de maneira integrada, ou seja, levando em consideração as interações

entre eles. A plataforma compreende a tecnologia de software central sobre a qual se dá a construção e manutenção do ECOS, por exemplo, Eclipse, Windows, SAP etc. (MANIKAS; HANSEN, 2013).

Um exemplo real de ECOS é o ambiente do iPhone, onde os atores são a empresa Apple, os usuários, os desenvolvedores da Apple, os desenvolvedores externos de aplicativos, e o iOS é a tecnologia de software central deste ECOS. Para compreender os processos envolvidos nos ECOSs, é necessário compreender sua estrutura, isto é, de que elementos são formados e o que significam para o ECOS.

Ativos de software são artefatos produzidos ou adquiridos e armazenados por uma organização (ADAMS; GOVEKAR, 2012). Portanto, ativos de ECOS representam os produtos do ECOS, podendo ser ativos de software (componentes, serviços, aplicações) e necessidades.

Uma plataforma de ECOS pode ter apoio de uma biblioteca de ativos, responsável por gerenciar o seu ciclo de vida. Uma biblioteca de ativos corresponde a um sistema de informação com mecanismos de publicação, documentação, armazenamento, busca e recuperação de ativos de software (SANTOS; WERNER, 2013).

Ativos de software, em geral, também podem ser considerados ativos reutilizáveis. Assim, ativos de ECOS englobam os ativos reutilizáveis, como componentes, serviços e até aplicações. Estes ativos reutilizáveis podem ser criados dentro da organização ou serem trazidos de fora da organização, ou mesmo do ECOS.

CONSIDERAÇÕES FINAIS

Chegamos ao final de mais uma unidade, na qual você conheceu o que é reutilização, componentes, *design patterns*, *framework*. Foram, também, apresentados, de forma sucinta, *Catalysis*, *UML Components* e *Kobra*. Reutilização é a capacidade de um item de software, previamente desenvolvido, ser usado novamente ou usado repetidamente em parte ou todo, com ou sem modificação (COOPER,1994).

Pode-se desenvolver com reuso (utiliza-se um item de software previamente desenvolvido) e, desenvolver para reuso (itens são desenvolvidos para torná-lo disponível para futuras reutilizações).

Os itens de software que se deseja desenvolver para reuso ou com reuso podem ser por meio de: componentes, *design patterns* ou frameworks. Um componente é uma unidade auto-contida que encapsula o estado e o comportamento de objetos. Ele especifica um contrato formal dos serviços por meio de interfaces providas e requeridas. Um *design pattern* nomeia, motiva e explica uma solução genérica a um problema recorrente que surge em uma situação específica. Frameworks criam uma infra-estrutura pré-fabricada para o desenvolvimento de aplicações de um domínio específico (MARTINS, 2005). Note, portanto, que quando se fala em *design pattern* a idéia é a de que os desenvolvedores de software experientes acumularam um repertório de princípios gerais e soluções que os guiavam frequentemente em suas decisões no desenvolvimento de novos softwares. Os frameworks provêm uma orientação arquitetural, definindo responsabilidades e colaborações entre objetos necessários para implementar uma determinada funcionalidade padrão.

Para finalizar, foram tratadas as linhas de produtos de software e ecossistemas. São abordagens que têm merecido atenção de estudiosos com ideias para reutilização em níveis diferentes do que é possibilitado por componentes, patterns e frameworks.

Assim, esperamos que com este aprendizado vocês possam tirar proveito da reutilização e, mais ainda, possa entregar produtos de melhor qualidade aos seus clientes.

ATIVIDADES



1. Segundo Sommerville (2011), as vantagens da reutilização ao desenvolvedor software são:

- I. Aumento da confiabilidade.
- II. Aumentam riscos do processo.
- III. Conformidade com padrões.
- IV. Aumenta o tempo de Desenvolvimento.

Assinale a alternativa correta:

- a) Todas estão corretas.
- b) Apenas I e II.
- c) Apenas I e III.
- d) Apenas I e IV.
- e) Apenas II e IV.

2. Tendo em vista as características dos *designs patterns*, é correto o que se afirma em:

- I. Fornecem um vocabulário comum aos desenvolvedores de software.
- II. Documentam a experiência existente e comprovada.
- III. Eles são únicos, não podem ser usados com outros padrões.
- IV. Descrevem e justificam soluções para problemas de sistemas aleatórios.

Assinale a alternativa correta:

- a) Todas estão corretas.
- b) Apenas I e II estão corretas.
- c) Apenas I e III estão corretas.
- d) Apenas II e III estão corretas.
- e) Apenas I e IV estão corretas.

ATIVIDADES



3. Em um *framework* pode-se identificar os *frozen spots* e os *hot spots*. Sobre seus conceitos, assinale a alternativa correta:
- a) *Hot spots*: constituem-se em partes do *framework* que são projetados para serem genéricos e, podem ser adaptados às necessidades da aplicação. Já *Frozen spots*: definem a arquitetura geral da aplicação, seus componentes e os relacionamentos entre eles. Eles permanecem fixos em todas as instâncias do *framework*.
 - b) *Hot spots*: constituem-se em partes do *framework* que são projetados para serem específicos e, podem ser adaptados às necessidades da aplicação . Já *Frozen spots*: definem a arquitetura geral da aplicação, seus componentes e os relacionamentos entre eles. Eles permanecem fixos em todas as instâncias do *framework*.
 - c) *Hot spots*: constituem-se em partes do *framework* que são projetados para serem genéricos e, não podem ser adaptados às necessidades da aplicação. Já *Frozen spots*: definem a arquitetura geral da aplicação, seus componentes e os relacionamentos entre eles. Eles permanecem fixos em todas as instâncias do *framework*.
 - d) *Hot spots*: constituem-se em partes do *framework* que são projetados para serem genéricos e, podem ser adaptados às necessidades da aplicação. Já *Frozen spots*: definem a arquitetura específica da aplicação, seus componentes e os relacionamentos entre eles. Eles permanecem fixos em todas as instâncias do *framework*.
 - e) *Hot spots*: constituem-se em partes do *framework* que são projetados para serem genéricos e, podem ser adaptados às necessidades da aplicação. Já *Frozen spots*: definem a arquitetura geral da aplicação, seus componentes e os relacionamentos entre eles. Eles são alterados em todas as instâncias do *framework*.

ATIVIDADES



4. Assinale a alternativa que apresenta a frase que é bastante usada para definir reutilização.
- a) Reutilização é a capacidade de um item de software, previamente desenvolvido, ser usado novamente ou usado repetidamente em parte ou todo, com ou sem modificação.
 - b) Reutilização é a capacidade de um item de software, ser usado novamente ou usado repetidamente em parte ou todo, com ou sem modificação.
 - c) Reutilização é a capacidade de um item de software, previamente desenvolvido, ser usado novamente ou usado repetidamente somente como um todo, sem modificação.
 - d) Reutilização é a capacidade de um item de software, ser usado novamente ou usado repetidamente em parte ou todo, sempre com modificação.
 - e) Reutilização é a capacidade de um item de software, sempre ser usado novamente, sem modificação.
5. Os itens de software que se deseja desenvolver para reuso ou com reuso podem ser por meio de: _____, _____ e _____.
- Assinale a alternativa que preenche corretamente a frase.
- a) Artefatos de projeto, Componentes e *Design patterns*.
 - b) Componentes, *Design patterns* e *Frameworks*.
 - c) *Frameworks*, Requisitos do cliente e *Design patterns*.
 - d) Artefatos de projeto, Componentes e *Frameworks*.
 - e) Requisitos do cliente, Componentes e *Design patterns*.



ECOS: A quarta geração do reúso de software

Desde a “Crise do Software”, no final da década de 1960, métodos, técnicas e ferramentas vêm sendo propostos para promover a qualidade em artefatos (especificações, modelos, códigos-fonte, entre outros), reduzindo o índice de falhas nos projetos de software. Já em 1969, Doug McIlroy propôs a ideia de decompor os sistemas de software em unidades reutilizáveis, chamadas componentes. No final dos anos 80, essa vertente da Engenharia de Software (ES) emergiu como uma área de pesquisa denominada Reutilização de Software: o uso de qualquer informação já disponível, por um desenvolvedor, no processo de criação de um sistema.

A primeira geração do reuso de software, nas décadas de 1970 e 1980, consolidou princípios de modularidade, tais como coesão e acoplamento, o que contribuiu para a construção de sistemas monolíticos compostos por rotinas que interagiam umas com as outras. As linguagens de programação orientadas a objetos impulsionaram a prática de modularização na programação estruturada, por meio de um paradigma orientado à modelagem de classes de objetos, descritos por características e comportamentos com base no mundo real, o chamado conceito da abstração.

Com a evolução da área de reutilização no final dos anos 80, técnicas como a Engenharia de Domínio e Arquitetura de Software viabilizaram a reutilização efetiva de componentes de aplicação por meio de frameworks e de linguagens de quarta geração. Na segunda geração do reuso, os engenheiros de software passaram a observar a experiência de outros setores da indústria na utilização de sistemas com base em componentes. A indústria de software percebia que muitos requisitos não eram novos, apesar de desenvolvidos num contexto um pouco diferente. Em paralelo, os ambientes de desenvolvimento (IDEs) estimularam um rápido crescimento do desenvolvimento orientado a objetos, com métodos para modelagem e construção que convergiam, na década de 1990, por conta do surgimento da UML (*Unified Modeling Language*).

Sistemas de linhas de produto são considerados como a terceira geração do reuso. Focando o desenvolvimento em infraestruturas web no final dos anos 1990 e mobile no início dos anos 2000, as pesquisas da ES ganham novos fatores de destaque: software crítico, gerência de dependências, serviços web, manutenção de sistemas legados, componentes de prateleira (*commercial-off-the-shelf*) e software livre (*open source*).

Para as organizações, estava claro o esforço para atingir os benefícios da reutilização, o que incluía focar no negócio, levando ao conceito de linhas de produto de software no final dos anos 90. Esse conceito considerava a construção de uma base de ativos reutilizáveis de uma organização levando em conta as similaridades entre os produtos para apoiar a sua variabilidade (reuso intraorganizacional). Entretanto, além do custo inherentte à construção dessa base de ativos, o desenvolvimento da infraestrutura (plataforma) também exigia investimentos, por envolver novos processos e ferramentas.



Com a adoção de linhas de produto de software dentro de uma organização e diante da evolução de seu escopo, as fronteiras começam a se expandir naturalmente, ou ao menos serem revistas. Assim, a plataforma passaria a estar disponível para partes externas à organização, ocorrendo, então, a transição para Ecossistemas de Software (ECOS), que são considerados a quarta geração do reuso de software.

Como poderia se supor, a expressão ECOS foi inspirada pelo conceito de ecossistemas biológicos, comerciais ou de negócios. Eles possuem basicamente três elementos: um centralizador (hub), uma plataforma (tecnologia) e um conjunto de agentes de nicho (*players*). O Google Play (Android) e o Apple Store (iOS) são exemplos de ECOS. Em essência, um conjunto de desenvolvedores externos é encorajado a utilizar uma plataforma pertencente a um grupo de organizações, e contribuir para o desenvolvimento de produtos.

Um ECOS também pode surgir da parceria entre empresas, por meio da integração de seus produtos, formando o chamado *system of systems*, ou ainda quando um desenvolvedor resolve mesclar diversas informações (*mashups*) de provedores distintos. Em todos os casos, percebem-se suas três dimensões: arquitetural (plataforma), social (atores) e de negócios. Quanto à última, vale notar que ela envolve conhecimento sobre o mercado, modelos de negócio, definição do portfólio de produtos, além da estratégia de licenças e de vendas.

Fonte: França (2015, on-line)¹.

MATERIAL COMPLEMENTAR



LIVRO

Padrões de Projeto: Soluções reutilizáveis de projeto de software orientado a objetos

Erich Gamma; Richard Helm; Ralph Johnson; John Vlissides

Editora: Bookman

Sinopse: reunindo uma grande experiência em software orientado a objetos, quatro projetistas de grande renome apresentam um catálogo de soluções simples e sucintas para os problemas mais frequentes na área de projeto. Os autores descrevem o que são padrões e como eles ajudam a projetar o software orientado a objetos. Prosseguem dando nomes, explicando e catalogando soluções de projeto. O leitor aprenderá como esses padrões se encaixam no desenvolvimento de software e o ajudam a resolver problemas de projeto de forma rápida e eficiente. São descritas as circunstâncias às quais se aplica cada padrão, quando ele pode ser usado e as consequências do seu uso no contexto maior do projeto. Todos os 23 padrões foram compilados de sistemas já existentes e estão baseados em exemplos reais, incluindo códigos que demonstram a sua implementação em linguagens orientadas a objetos, como C++ e Smalltalk.

Comentário: esta é a versão traduzida do livro Design patterns originalmente escrito em inglês.



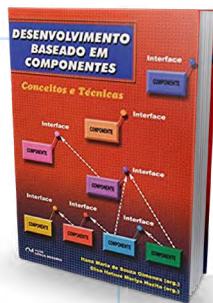
LIVRO

Desenvolvimento Baseado em Componentes: Conceitos e Técnicas

Itana Gimenes e Elisa Huzita

Editora: Ciência Moderna

Sinopse: a tecnologia de componentes é discutida amplamente, sob vários aspectos, assim como o processo de seu desenvolvimento, o qual é conceituado à luz da literatura e das tecnologias atuais pelo livro. 'Desenvolvimento Baseado em Componentes' é adequado para servir de referência para cursos relacionados à Engenharia de Software, uma vez que discute tudo o que já foi anteriormente citado e a re-engenharia de software, a confiabilidade de todo o processo no contexto da nova tecnologia; as relações entre componentes e padrões de projeto e frameworks. O livro traz - Conceitos Básicos; Desenvolvimento Baseado em Componentes; A Engenharia de Domínio e o Desenvolvimento; Reengenharia de Software Baseada em Componentes; Desenvolvimento Baseado em Componentes e confiabilidade; Desenvolvimento Baseado em Componentes e Padrões; Software Baseado em Componentes; Uma Revisão Sobre Teste e conclusões.



NA WEB

Sugiro que você conheça por meio do link o trabalho dos autores Leonardo Gresta Paulino Murta e Cláudia Maria Lima Werner que apresenta uma infraestrutura integrada de Gerência de Configuração de Software (GCS) para o Desenvolvimento Baseado em Componentes (DBC). Para saber mais, acesse: <www2.ic.uff.br/~leomurta/papers/murta2007.pdf>.



REFERÊNCIAS

- ADAMS, P.; GOVEKAR, M. **Hype Cycle for IT Operations Management.** Gartner Technical Professional Advice, 2012.
- ALEXANDER, C.; ISHIKAWA, S., SILVERSTEIN, M.; JACOBSON, M.; FIKSDAHL-KING, I.; ANGEL, S. **A Pattern Language.** New York: Oxford University Press, 1977.
- ARANGO, G. **Domain engineering for software reuse.** Irvine: University of California, UCI-ICS, 1988, 195f.
- ATKINSON,C.; BAYER,J.; LAITENBERGER, O.; ZETTEL,J. Component based software engineering: The Kobra approach. In: **International Workshop on Component Based Software Engineering**, 2000.
- BARROCA, L.; HALL,J.; HALL,P. **Software architectures, advances and applications.** 1 ed. Alemanha: Springer Verlag, 2000.
- BASS, L.; CLEMENTS, P.; KAZMAN, R. **Software Architecture in Practice**, 2. ed. Boston: Addison-Wesley, 2003.
- BLOIS, A. P. Uma Abordagem de Projeto Arquitetural baseado em Componentes no contexto de Engenharia de Domínio. 2006. Tese (Doutorado em Engenharia de Sistemas e Computação) - Universidade Federal do Rio de Janeiro, 2006.
- BOSCH, J. From Software Product Lines to Software Ecosystem. In: **Proceedings of the 13th International Software Product Line Conference**. San Francisco, USA, pp. 1-10, 2009.
- BUSCHMANN, F. at al. **Pattern-Oriented Software Architecture:** A System of Patterns. v.1. Great Britain: Wiley, 1996. 487p.
- CHEESMAN, J.; DANIELS, J. **UML Components, a simple process for specifying component based software.** 1 ed. Reading: Addison-Wesley, 2001.
- CLEMENTS, P.; NORTHRUP, L. **Software Product Lines:** Practices and Patterns. Addison Wesley, 2002.
- COOPER, J. Reuse business implications. In: **Encyclopedia of software engineering**. Nova York: John Wiley & Sons, 1994.
- DESCHAMPS, F. **Padrões de Projeto. Uma Introdução.** Notas de Aula. Departamento de Automação e Sistemas (DAS). Universidade Federal de Santa Catarina, 2005.
- D'SOUZA, D.; WILLS, A. **Objects, Components and Frameworks:** The Catalysis Approach. 1 ed. EUA; Addison Wesley Publishing Company, 1998.
- FAYAD, M. **Building Application frameworks:** Object-oriented foundations of framework design. 1.ed. Nova York: John Wiley & Sons, 1999.
- FREEMAN, P. Reusable software engineering: concepts and research directions. In: **Software Reusability (tutorial)**. Washington, D. C.IEE Computer Society Press. 1987.

REFERÊNCIAS

- GAMMA, E. et al. **Design patterns:** Elements of Reusable Object-Oriented Software. Addison Wesley, 22 ed., 2002, United States of America.
- GIMENES, I. M. S.; TRAVASSOS, G. H. O Enfoque de Linha de Produto para Desenvolvimento de Software. In: Ingrid Jansch Porto. (Org.). **XXI Jornada de Atualização em Informática (JAI)** - Livro Texto. 1. ed. Porto Alegre: Sociedade Brasileira de Computação, 2002, v. 2, p. 01-31.
- GIMENES, I. M. S; HUZITA, E. H. M. **Desenvolvimento Baseado em Componentes:** Conceitos e Técnicas. Rio de Janeiro. Editora Ciência Moderna, 2005.
- KRUEGER, C. W. **Software Reuse.** ACM Computing Surveys. Nova York. V. 24, n. 2, junho 1992, p. 83-131.
- LEWIS, T. G. **Object Oriented frameworks.** Connecticut Manning Publications Company, 1995.
- MANIKAS, K.; HANSEN, K. M. (2013) Software ecosystems - A systematic literature review. **Journal of Systems and Software**, v. 86, n. 5, maio, 2013, p. 1294-1306.
- MARTINS, E. **Reutilização de Software.** Notas de aula. Instituto de Computação. Universidade de Campinas, 2005.
- NAKAGAWA, E. Y. **Uma contribuição ao projeto arquitetural de ambientes de engenharia de software.** 2006. 252 f. Tese (Doutorado) - Universidade de São Paulo, São Carlos 2006.
- PRIETRO-DIAS, R. **Software reusability.** 1.ed. Crystal City. Ellis Horwood, 1994.
- RUMBAUGH, J.; JACOBSON, I.; BOOCHE, G. **The unified language reference manual.** 1 ed. Reading: Addison-Wesley, 1998.
- SANTOS, R. P.; WERNER, C. On the Impact of Software Ecosystems in Requirements Communication and Management. In: **Proceedings of the ER@BR, IEEE International Conference on Requirements Engineering.** Rio de Janeiro, 2013, p. 190-195.
- SAMETINGER, J. **Software Engineering with reusable components.** Springer Verlag Berlin Heldelberg, 1997.
- SHAW, M.; GARLAN, D. **Software architecture:** Perspectives on an emerging discipline. 1 ed. EUA: Prentice Hall, 1996.
- SOMMERVILLE, I. **Engenharia de Software.** 9. ed. São Paulo: Pearson Prentice Hall, 2011.
- WALLNAU, K.; HISSAM, S; SEACORD, S. **Building systems from commercial components.** 1 ed. Reading Addison-Wesley Publishing, 2001.
- WARMER,J.; KLEPPE, A. **Object Constraint language:** precise modeling with UML. Addison-Wesley Object Technology Series, 1998.



REFERÊNCIAS

REFERÊNCIA ON-LINE

¹Em: <<https://www.ibm.com/developerworks/community/blogs/tlcbr/entry/mp227?lang=en>>. Acesso em: 9 maio 2018.



GABARITO

1. Alternativa C.

2. Alternativa B.

3. Alternativa A.

4. Alternativa A.

5. Alternativa B.



REFATORAÇÃO PARA PADRÕES

Objetivos de Aprendizagem

- Estudar os conceitos e princípios da Refatoração.
- Conhecer o Catálogo de Refatoração para Padrões.
- Entender o Catálogo de Padrões de Projeto e qual a sua relação com a Refatoração.

Plano de Estudo

A seguir, apresentam-se os tópicos que você estudará nesta unidade:

- Refatoração
- Catálogo de Refatoração para Padrões
- O Catálogo de Padrões de Projeto

INTRODUÇÃO

Olá, aluno(a)! Esta unidade tem por objetivo introduzir conceitos e princípios sobre a refatoração para padrões e como ela pode ser aplicada de maneira controlada e eficiente, que não introduza erros no código e que melhore metodicamente a sua estrutura.

Iniciaremos com uma introdução sobre a Refatoração, seus conceitos, definições e sua história, que iniciou com Martin Fowler (1999), popularizando o termo refatoração para referir-se à atividade de aprimoramento de seu estilo de código sem alterar o seu comportamento. O termo é usado para cada uma das mudanças individuais que você pode efetuar para melhorar a estrutura de seu código. Fowler define um catálogo de sintomas indicando que o seu código precisa de refatoração (o que ele chama de “maus cheiros”) e fornece um catálogo de refatoração úteis.

Em seguida, conheceremos os motivos para refatorar um código e ainda alguns problemas que podem ocorrer com a Refatoração. E para resolver esses problemas, veremos como encontrar os maus cheiros no código e a limpá-los com a Refatoração.

Estudaremos o catálogo de refatorações com alguns exemplos de utilização, já que Fowler explica que não é um catálogo definitivo, é apenas o começo. Também falaremos sobre a Refatoração para padrões e como a utilização de padrões pode melhorar o seu código.

Entenderemos porque devemos manter um código limpo e como a refatoração ajuda neste processo, já que o propósito da refatoração é tornar o software mais fácil de entender e modificar. Você pode fazer muitas mudanças no software que alterem pouca coisa ou nada no comportamento observável. Somente as mudanças feitas para tornar o software mais fácil de entender são consideradas refatorações.

Também estudaremos as técnicas de refatoração, os seus padrões e como são divididas em categorias. Serão mostradas algumas dessas técnicas de refatoração do catálogo de Fowler (2004) agrupadas por categorias.

Preparado(a) para começar a leitura? Então, vamos seguir em frente. Bons estudos.

Refactoring

REFATORAÇÃO

Para Kerievsky (2008, p. 35):

[...] uma refatoração é uma transformação que preserva comportamento” ou segundo Fowler (2004, p. 52) uma refatoração é “uma alteração feita na estrutura interna do software para torná-lo mais fácil de ser entendido e menos custoso de ser modificado sem alterar seu comportamento observável.

Refatoração é a mudança de um código fonte, na estrutura interna do software, visando melhorar o entendimento e a manutenibilidade sem alterar seu comportamento e suas funções externas. A refatoração surgiu quando alguns desenvolvedores foram analisar seus códigos para alterar ou incluir novas funcionalidades, e notaram que os códigos já existentes estavam em grande parte desestruturados, trechos repetidos e de difícil compreensão e manutenção.

O processo de refatoração envolve remover códigos duplicados, simplificação de lógica condicional e deixar os códigos mais claros e limpos. Refatorar um código, significa modificá-lo incansavelmente para melhorá-lo. E esta modificação para melhorá-lo, pode envolver algo pequeno como trocar o nome de uma variável ou algo grande, como unificar duas hierarquias. (KERIEVSKY, 2008).

Refatorar o código em pequenos passos ou blocos, ajuda a prevenir a introdução de defeitos. Algumas refatorações podem levar minutos para serem executadas e outras requerem um esforço sustentado por um tempo maior (dias, semanas ou meses). E mesmo as que requerem um tempo maior, são implementadas em pequenos passos. E conforme Kerievsky (2008, p. 36) “é melhor refatorar continuamente, ao invés de em fases. Quando você vê código que necessita de melhorias, melhore-o”.

Tsui e Karan (2013, p. 141) explica a respeito da refatoração:

Martin Fowler (1999) popularizou o termo *refatoração* para referir-se à atividade de aprimoramento de seu estilo de código sem alterar o seu comportamento. Ele também utiliza este termo para cada uma das mudanças individuais que você pode efetuar para melhorar a estrutura de seu código. Fowler define um catálogo de sintomas indicando que o seu código precisa de refatoração (o que ele chama de “maus cheiros”) e fornece um catálogo de refatoração úteis.

Mas por que refatorar? Existem muitas razões para refatorar um código. Veja algumas das motivações mais comuns (tabela 1):

Tabela 1 – Razões para refatorar um código

Refatorar melhora o projeto do Software	Sem refatoração, o projeto do programa termina por se deteriorar. A medida que se altera o código, o código se desestrutura. Refatoração é como arrumar o código. A refatoração sistemática ajuda o código a conservar sua forma.
Refatorar torna o software mais fácil de entender	Programar é de certa forma conversar com o seu computador. Você escreve o código que diz o que fazer e ele responde fazendo exatamente o que você lhe diz. Todavia há outros usuários do seu código fonte. Alguém futuramente lendo seu código para fazer alterações. A refatoração ajuda a tornar seu código mais legível. Muitos programadores usam a refatoração para entender os códigos que não estão familiarizados.
Refatorar ajuda a encontrar falhas	A melhora no entendimento do código também ajuda a localizar falhas. Quando se refatora o código, se comprehende o que ele faz e pode-se aplicar o conhecimento adquirido de volta no código. Com a estrutura mais clara, fica mais fácil achar falhas. Refatorar ajuda o programador a ser muito mais efetivo na escrita de um código robusto.
Refatorar ajuda a programar mais rapidamente	Todos os fatores anteriores levam a este. Refatoração melhora a qualidade, melhora o projeto, melhora a legibilidade, reduz as falhas. Com um bom projeto é essencial na manutenção da velocidade de desenvolvimento do software. Refatorar ajuda você a desenvolver software mais rapidamente, porque evita que o projeto do sistema se deteriore.

Fonte: adaptado Fowler (2004, p. 56).

Na refatoração para obter os melhores resultados, é necessário que seu código seja olhado por muitos. Segundo Kerievsky (2008, p. 39) “esta é uma das razões pelas quais a programação extrema sugere as práticas de programação em pares e propriedades coletivas de código”.

CÓDIGO LEGÍVEL POR HUMANO

Ao escrever códigos, devemos produzir um método que seja legível por humanos. Para exemplificar, seguem dois códigos que produzem a mesma data:

Código 1:

```
java.util.Calendar c = java.util.Calendar.getInstance();
c.set(2005, java.util.Calendar.DECEMBER, 10);
c.getTime()
```

Código 2:

```
december (10, 2017)
Esse código chama o seguinte método:
public Date december (int day, int year)
```

Figura 1 - Código legível por humano

Fonte: a autora.

Se os códigos produzem a mesma data, qual a diferença entre eles? O código 2 pode ser lido como uma linguagem falada e está separando o código importante do código que distrai. Para Fowler (2004, p. 29) “qualquer tolo consegue escrever código que um computador entenda. Bons programadores escrevem código que humanos possam entender”.

Por isso, mantenha sempre o código limpo. E para mantê-lo limpo, devemos remover códigos duplicados, deixá-los simplificados e claros. Código limpo, ajuda no desenvolvimento rápido, que leva a clientes e programadores felizes e ao sucesso do sistema. Em desenvolvimento ágil, se faz necessária a refatoração

para deixar o código sempre limpo, pois sem refatoração as mudanças no código tornam-se cada vez mais difíceis e caras. A refatoração pode melhorar a qualidade do código e facilitar as mudanças no sistema, além de ser considerada uma das técnicas mais poderosas para a produção de um bom código.

Você deve estar pensando: a refatoração é uma limpeza de código? Segundo Fowler (2004), a resposta é sim, pois refatoração fornece uma técnica para limpar o código de maneira mais eficiente e controlada. Para explicar esta resposta, o autor comenta:

[...] primeiro, o propósito da refatoração é tornar o software mais fácil de entender e modificar. Você pode fazer muitas mudanças no software que alterem pouca coisa ou nada no comportamento observável. Somente as mudanças feitas para tornar o software mais fácil de entender são consideradas refatorações. Um bom contraste é a otimização de desempenho. Assim como a refatoração, a otimização de desempenho geralmente não altera o comportamento de um componente (além da sua velocidade), ela altera apenas a estrutura interna. Entretanto, o propósito é diferente. A otimização de desempenho muitas vezes torna o código mais difícil de entender, mas você precisa fazê-la para obter o desempenho de que necessita. A segunda coisa que quero destacar é que a refatoração não altera o comportamento observável do software. Esse ainda executa a mesma função de antes. Qualquer usuário, seja ele um usuário final ou outro programador, não é capaz de dizer que algo mudou (FOWLER, 2004, p. 53).

Portanto:

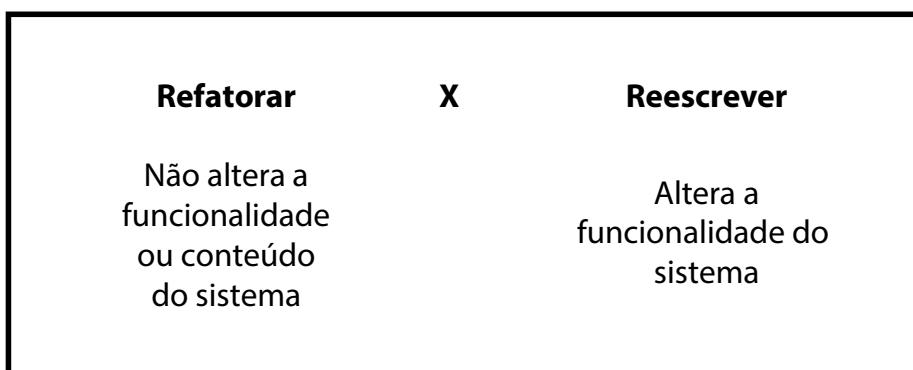


Figura 2 – Refatorar x Reescrever

Fonte: a autora.



SAIBA MAIS

Metáfora de Kent Beck dos dois chapéus

Quando usa refatoração para desenvolver software, você divide seu tempo entre duas atividades distintas: adicionar funcionalidades e refatorar. Quando adiciona funcionalidades, você não deve alterar código preexistente. Você está apenas adicionando novas capacidades. Você pode medir seu progresso adicionando testes e os executando. Quando você refatora, faz questão de não acrescentar novas funcionalidades; apenas reestruturar o código. À medida que você desenvolve o software, provavelmente se descobre trocando chapéus com frequência. Você começa tentando adicionar uma nova função e percebe que seria muito fácil se o código estivesse estruturado de forma diferente. Então você troca de chapéu e refatora durante um certo período. Assim que o código estiver mais bem estruturado, você troca de chapéu e acrescenta a nova função. Assim que você tenha a nova função ativa, percebe que a codificou de uma maneira difícil de entender e então troca de chapéu novamente e refatora. Tudo isso pode levar apenas dez minutos, mas durante este tempo você deve estar ciente de qual chapéu está usando.

Fonte: Fowler (2004, p. 53).

QUANDO REFATORAR?

A refatoração pode surgir em dois momentos: para melhorar o código existente ou quando temos que jogar fora e começar do zero. E a empresa tem a responsabilidade de avaliar a situação e decidir quando é a hora de optar por um ou por outro. Todavia, segundo Fowler (2004, pg. 56), três vezes você refatora.

- **Refatore quando acrescentar funções:** o mais comum é refatorar quando se adiciona uma nova característica ao software e é preciso compreender o código para que possa modificá-lo. Outro motivo para refatorar é quando o projeto não ajuda a acrescentar a nova funcionalidade facilmente. Uma vez refatorado o código, fica mais fácil, tranquilo e rápido acrescentar novas características.

- **Refatore quando precisar consertar uma falha:** uma das utilidades de refatorar é tornar o código mais compreensível. Com o código mais compreensível, fica mais fácil de encontrar a falha, pois o código não estava claro o suficiente para que percebesse a falha.
- **Refatore enquanto revisa o código:** revisar o código ajuda a equipe a conhecer o código. Elas são importantes na escrita de um código claro. Refatorar também ajuda a revisar o código de outras pessoas e a gerar resultados mais concretos.

REFLITA



Se você quiser refatorar, a pré-condição essencial é ter testes sólidos. Escrever bons testes aumenta a velocidade de programação, mesmo se não estiver refatorando.

(Fowler)

“MAUS CHEIROS” NO CÓDIGO

O fato de você saber como a refatoração funciona, não significa que sabe quando usá-la. Decidir quando parar a refatoração é tão importante quanto saber a sua mecânica. Para isso, temos um catálogo de “maus cheiros”, para ser usado quando não estiver certo sobre a refatoração a ser feita. Este catálogo ajuda a identificar o que o desenvolvedor está sentido e ajuda a identificar e detectar o mau cheiro no código e lhe apontar a direção certa, quando for realizar a refatoração.

Mas por que o termo “maus cheiros”? É uma metáfora usada para descrever sistemas de software desenvolvidos com práticas de programação ruim e que podem ser resolvidos por meio da refatoração.

A seguir é mostrado o catálogo de “maus cheiros” fornecido por Fowler (tabela 2):

Tabela 2 - Catálogo de Maus Cheiros

Código duplicado mostrando desperdício

Este cheiro é o número 1 do ranking dos maus cheiros. Se você vir o mesmo código em mais de um lugar, ele está duplicado e com certeza seu sistema ficará melhor se você encontrar um meio de unificá-los. O problema mais simples de código duplicado é quando você tem a mesma expressão em dois métodos da mesma classe. Outro problema é quando você tem a mesma expressão em duas subclasses irmãs.

Método longo

Os programas orientados a objetos vivem melhor e por mais tempo quando possuem métodos curtos. Quanto maior for o procedimento, mais difícil é entendê-lo. O importante não é o tamanho do método, mas a distância semântica entre o que o método faz e como ele faz.

Classe grande

Quando uma classe tenta fazer muita coisa, ela frequentemente tem variáveis de instância demais. Quando uma classe tem variáveis de instância em excesso, o código duplicado não deve estar longe. Do mesmo modo que com uma classe com variáveis de instância em excesso uma classe com código demais é solo fértil para código duplicado, caos e morte.

Lista de parâmetros longa

São difíceis de entender, porque se tornam inconsistentes e difíceis de usar e porque você irá sempre alterá-las à medida que precisar de mais dados. Se a lista de parâmetros for longa demais ou as alterações frequentes demais, você precisa repensar sua estrutura de dependência.

Alteração divergente

Estruturamos nosso software para tornar as alterações mais fáceis. Quando temos que fazer uma alteração, queremos ser capazes de ir a um único ponto do sistema, claramente definido, e fazer essa alteração. Quando você não consegue fazer isso, está sentindo o cheiro de dois problemas intimamente relacionados. Uma alteração divergente ocorre quando uma classe é frequentemente alterada de diferentes maneiras por diferentes razões. Qualquer alteração para lidar com uma variação deve alterar uma única classe, e toda a digitação na nova classe deve expressar a variação.

Cirurgia com rifle

Ela é parecida com a alteração divergente, mas é o oposto. Você faz uma cirurgia com rifle quando, cada vez que executa uma mudança, tem que fazer muitas alterações pequenas em muitas classes diferentes. Quando as alterações estão por toda parte, elas são difíceis de encontrar e é fácil deixar de fazer alguma alteração importante. Alteração divergente é uma classe que sofre com muitos tipos de alterações e a cirurgia com rifle é uma mudança que altera muitas classes.

Inveja dos dados

A essência dos objetos é que eles são uma técnica para empacotar dados com os processos usados nestes dados. Um indício clássico de problema é um método que parece mais interessado em uma classe diferente daquela na qual ele se encontra. O foco mais comum da inveja são os dados. Método que chama meia dúzia de métodos de leitura em outro objeto para calcular algum valor, ou seja, o método claramente quer estar em algum outro lugar. Às vezes, somente parte do método sofre de inveja. A regra fundamental é colocar juntas as coisas que são alteradas juntas. O dado e o comportamento que o referencia normalmente e são alterados juntos, mas há exceções. Quando essas exceções ocorrem, movemos o comportamento para manter as alterações em um único local.

Grupos de dados

Os dados tendem a ser como crianças, gostam de ficar em grupos. Agrupamento de dados que perambulam juntos, na verdade, deveriam ser criados no seu próprio objeto. Reduzir listas de campos e de parâmetros irá remover alguns dos maus cheiros. Um bom teste é considerar apagar um dos valores dos dados se você fizesse isso, os outros fariam sentidos? Caso não é um sinal seguro de que você tem um objeto que está louco para nascer.

Comandos Switch

Um dos fenômenos mais óbvios do código orientado a objetos é sua comparativa falta de comandos *switch* (ou *case*). O problema com os comandos *switch* é essencialmente o da duplicação. Muitas vezes você encontra o mesmo comando *switch* espalhado por diversos lugares do mesmo programa. Se acrescentar uma nova cláusula ao *switch*, você tem que encontrar todos esses comandos *switch* e alterá-los. O conceito de polimorfismo da orientação a objetos lhe dá uma maneira elegante de lidar com esse problema.

Hierarquias paralelas de herança

Este é um caso especial de cirurgia com rifle. Neste caso, cada vez que você cria uma subclasse de uma determinada classe, tem também que criar uma subclasse de outra. Você pode reconhecer este cheiro porque os prefixos dos nomes das classes em uma hierarquia são os mesmos dos da hierarquia da outra. A estratégia geral para eliminar a duplicação é se assegurar de que instâncias de uma hierarquia se referem a instâncias da outra.

Classe ociosa

Cada classe que você cria custa dinheiro para manter e compreender. Uma classe que não esteja fazendo o suficiente para se pagar deve ser eliminada. Muitas vezes ela pode ser uma classe que costumava se pagar, mas que foi reduzida com a refatoração. Pode ser também uma classe que foi acrescentada devido a alterações que foram planejadas, mas não executadas. De qualquer modo, deixe a classe morrer com dignidade.

Generalidade Especulativa

A generalidade especulativa pode ser identificada quando os únicos usuários de um método ou classe formam os casos de teste. Se você encontrar um método ou uma classe assim, apague tanto o método quanto o caso de testes que o exerce. Se você tem um método ou uma classe que é um auxiliar em um caso de teste que exerce uma funcionalidade legítima, você tem que deixá-lo.

Campo Temporário

Às vezes você encontra um objeto no qual uma variável de instância recebe um valor apenas em determinadas circunstâncias. Tal código é difícil de entender, porque você espera que um objeto precise de todas as suas variáveis. Tentar entender porque uma variável está lá quando não parece ser usada pode lhe enlouquecer.

Cadeias de Mensagens

Você se depara com cadeias de mensagens quando um cliente pede um objeto para outro objeto, ao qual o cliente pede então outro objeto, ao qual o cliente pede então outro objeto e assim por diante. Navegar dessa maneira significa que o cliente está acoplado à estrutura de navegação. Qualquer mudança nos relacionamentos intermediários obriga a uma mudança no cliente.

Intermediário

Uma das características básicas da orientação a objetos é o encapsulamento – ocultar detalhes internos do resto do mundo. O encapsulamento frequentemente vem junto com delegação. Você olha uma interface de uma classe e descobre que metade dos métodos delegam para outra classe.

Intimidade inadequada

Às vezes as classes se tornam íntimas demais e gastam tempo demais sondando as partes privadas das outras. Classes íntimas demais precisam ser separadas. A herança pode muitas vezes levar à intimidade excessiva e as subclasses sempre saberão mais sobre seus pais do que esses gostariam que elas soubessem.

Herança recusada

Subclasses conseguem herdar métodos e dados de seus pais, mas e se não quiserem ou precisarem do que lhes é dado? Elas recebem todos estes ótimos presentes e escolhem apenas alguns para brincar. A história tradicional é que isso significa que a hierarquia está errada.

Comentários

Não se preocupe, pode usar comentários no seu código, eles não são cheiros ruins. São considerados cheiro doce, pois muitas vezes são usados como desodorantes. Pois é com muita frequência que se vê códigos cheios de comentários e percebe que eles estão lá porque o código é ruim.

Fonte: adaptado de Fowler, 2004 (p. 71-80).

Para Tsui (2013, p. 142) “qualquer um destes sintomas, bem como os outros que Fowler cita e aqueles que você desenvolverá, indicará que seu código pode ser melhorado. Você pode utilizar refatoração para ajudá-lo a lidar com estes problemas”.

Apesar de trazer benefício para um código fonte existente, a refatoração pode apresentar riscos se aplicada da forma errada, como atraso do projeto, introdução de falhas no sistema, tornar o código ilegível e não modificável, portanto é uma mudança que deve ser efetuada com cuidado.



SAIBA MAIS

Um dos problemas no desenvolvimento de software reutilizável é que, frequentemente, o software tem que ser reorganizado ou refatorado. Os padrões de projeto ajudam a determinar como reorganizar um projeto e podem reduzir o volume de refatoração que você terá que fazer mais tarde. O ciclo de vida do software orientado a objetos tem várias fases. A fase de prototipação é uma atividade agitada à medida que o software é trazido à vida por meio da prototipação rápida em mudanças incrementais, até que satisfaça um conjunto inicial de requisitos e atinja a adolescência.

Fonte: Gamma (2007, p. 325)

CATÁLOGO DE REFATORAÇÃO PARA PADRÕES

Neste tópico será descrito um catálogo inicial de refatorações, que segundo Fowler (2004, p. 95) “este catálogo não é de forma alguma completo ou fechado, mas deve fornecer uma base sólida para o seu próprio trabalho de refatoração”. Vamos começar falando sobre o formato de refatorações criado por Fowler.

FORMATO DE REFATORAÇÕES

Trata-se de um formato padrão, em que cada refatoração representa cinco partes, conforme a tabela a seguir:

Tabela 3 – Formato das Refatorações

NOME	O nome é importante para construir um vocabulário de refatorações.
RESUMO	Breve resumo da situação atual na qual precisa da refatoração e um resumo do que ela faz. O resumo ajuda a refatorar mais rapidamente.
MOTIVAÇÃO	Descreve porque a refatoração deve ser feita e as circunstâncias nas quais não deve ser feita.
MECÂNICA	É uma descrição concisa, passo a passo, de como executar a refatoração.
EXEMPLOS	Mostra um uso bem simples da refatoração para ilustrar como ela funciona.

Fonte: adaptado de Fowler (2004, p. 95).



Um padrão de projeto nomeia, abstrai e identifica os aspectos-chave de uma estrutura de projeto comum para torná-la útil para a criação de um projeto orientado a objetos reutilizáveis. O padrão de projeto identifica as classes e instâncias participantes, seus papéis, colaborações e a distribuição de responsabilidades. Cada padrão de projeto focaliza um problema ou tópico particular de projeto orientado a objetos. Ele descreve em que situação pode ser aplicado, se ele pode ser aplicado em função de outras restrições de projeto e as consequências, custos e benefícios de sua utilização.

Fonte: Gamma (2007, p. 20).

CATÁLOGO DE TÉCNICAS DE REFATORAÇÃO

As técnicas de refatoração, conforme Fowler (2004), são diferentes padrões que se dividem em categorias. A seguir serão apresentadas algumas dessas técnicas de refatoração do catálogo de Fowler (2004) agrupadas por categorias.

Compondo Métodos

Segundo Fowler (2004), esta categoria agrupa técnicas que são utilizadas para a composição de métodos para empacotar códigos apropriadamente, ajudando a reduzir o acoplamento e a sua complexidade. Quase sempre os métodos que causam problemas são os métodos longos demais, porque frequentemente eles contêm muitas informações escondidas pela complexidade do código. A seguir (figura 3), as técnicas desta categoria:

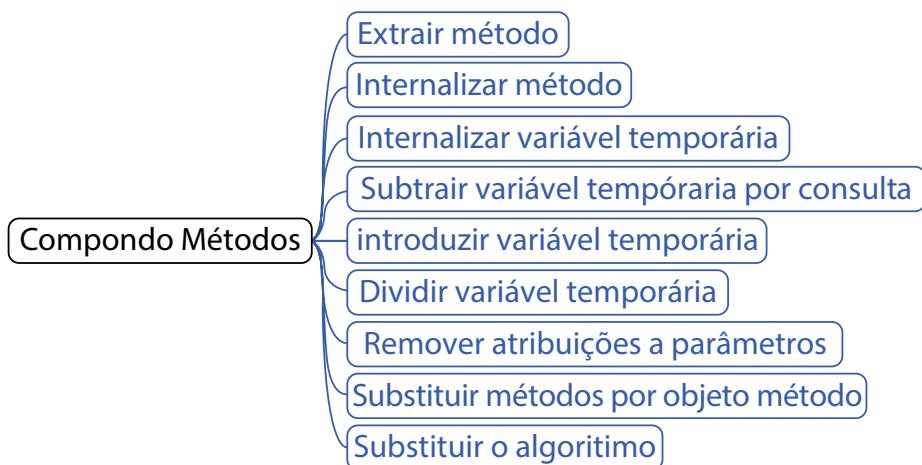


Figura 3 – Categoria Compondo Métodos

Fonte: adaptado de Fowler (2004).

Extrair Método

Extrair método constitui-se em dividir o método grande em pequenos fragmentos. Feito isso, transforme o fragmento em um método com nomes apropriados que expliquem seu propósito, para que tenha maior legibilidade para seus desenvolvedores (FOWLER, 2004).

```

void imprimirDivida(double quantia) {
    imprimirCabeçalho();

    //imprimir os detalhes

    System.out.println("nome:" + _nome);
    System.out.println("quantia:" + quantia);
}

```



```

void imprimirDivida(double quantia) {
    imprimirCabeçalho();
    imprimirDetalhes(quantia);
}

private void imprimirDetalhe(double quantia) {
    System.out.println("nome:" + _nome);
    System.out.println("quantia:" + quantia);
}

```

Figura 4 – Exemplo de técnica *Extrair Método*

Fonte: adaptado de Fowler (2004, p. 100).

Internalizar Método

Internalizar método é colocar o corpo do método dentro do corpo do que o chama e remover o método (FOWLER, 2004).

```

int lerAvaliação () {
    return (maisDeCincoEntregasAtrasadas ()) ? 2: 1;
}
boolean maisDeCincoEntregasAtrasadas () {
    return _numeroDeEntregasAtrasadas >5;
}

```



```

int lerAvaliação () {
    return (_maisDeCincoEntregasAtrasadas >5 )? 2: 1;
}

```

Figura 5 – Exemplo de técnica *Internalizar Método*

Fonte: adaptado de Fowler (2004, p. 106).

Internalizar Variável Temporária

Se você tem uma variável temporária que recebe uma única atribuição de uma expressão simples, substitua todas as referências a essa temporária pela expressão (FOWLER, 2004).

```
double precoBase = umPedido.precobase ();
return (precoBase > 1000);
}
```



```
return (umPedido.precioBase () > 1000);
```

Figura 6 – Exemplo de técnica *Internalizar Variável Temporária*

Fonte: adaptado de Fowler (2004, p. 107).

Substituir Algoritmo

Utiliza-se esta técnica quando se quer substituir um algoritmo por um mais claro. Substitua o corpo do método pelo novo algoritmo (FOWLER, 2004).

```
String pessoaEncontrada (String[] pessoas) {
    for ( int i = 0; 1 < pessoas.length; i++) {
        if (pessoas [i].equals ("Don")){
            return "Don";
        }
        if (pessoas [i].equals ("Jhon")){
            return "Jhon";
        }
        if (pessoas [i].equals ("Kent")){
            return "Kent";
        }
    }
    return "";
}
```



```
String pessoaEncontrada (String[] pessoas){
    List candidatos = Arrays.asList (new String [] {"Don", "Jhon", "kent"});
    for (int i =0; i<pessoas.length; i++) {
        if (candidatos.contains (pessoas[i]))
            return pessoas [i];
    }
    return "";
}
```

Figura 7 – Exemplo de técnica *Substituir o Algoritmo*

Fonte: adaptado de Fowler (2004, p. 123).

Movendo Recursos Entre Objetos

Segundo Fowler (2004, p. 123) “uma das decisões mais fundamentais, no projeto orientado a objetos é onde colocar as responsabilidades”. E a refatoração é usada para redefinir essas decisões fundamentais. A seguir as técnicas que podem ser usadas nesta categoria.

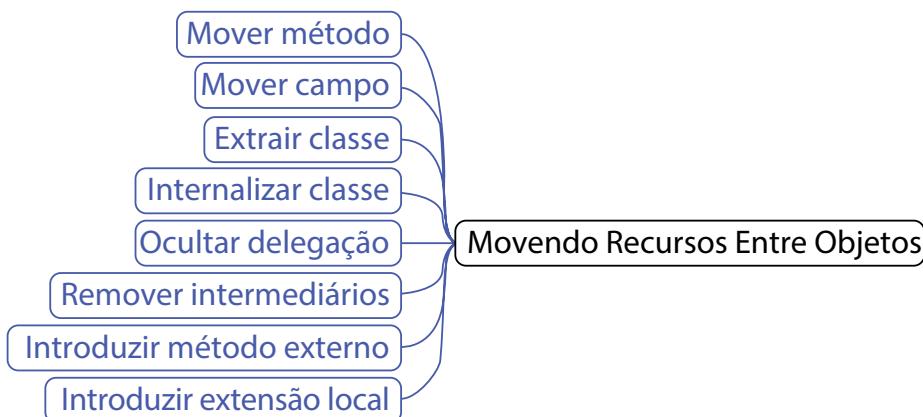


Figura 8 – Categoria Movendo Recurso Entre Objetos

Fonte: adaptado de Fowler (2004).

Reprodução proibida. Art. 184 do Código Penal e Lei 9.510 de 19 de fevereiro de 1998.

Mover método

Se um método está usando ou sendo usado por mais recursos de outra classe do que a classe na qual ele está definido, crie um novo método com o corpo similar e transforme o método antigo em uma simples delegação ou remova completamente (FOWLER, 2004).

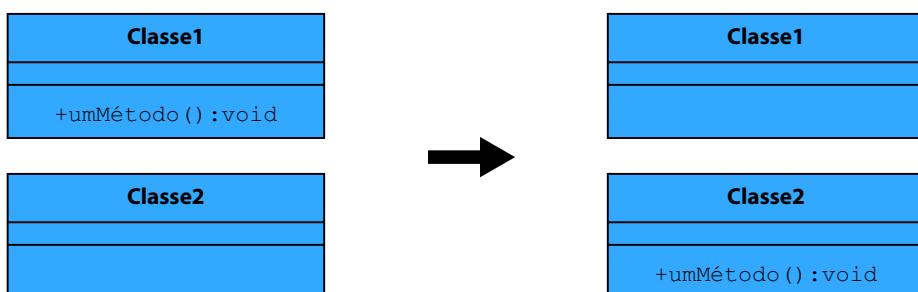


Figura 9 – Técnica Mover Método

Fonte: adaptado de Fowler (2004).

Mover campo

Segundo Fowler (2004, p. 129) “um campo é, ou será, usado por outra classe mais do que a classe na qual ele está definido”. Nesta técnica, crie um novo campo na classe alvo e altere todos os usuários deste campo.

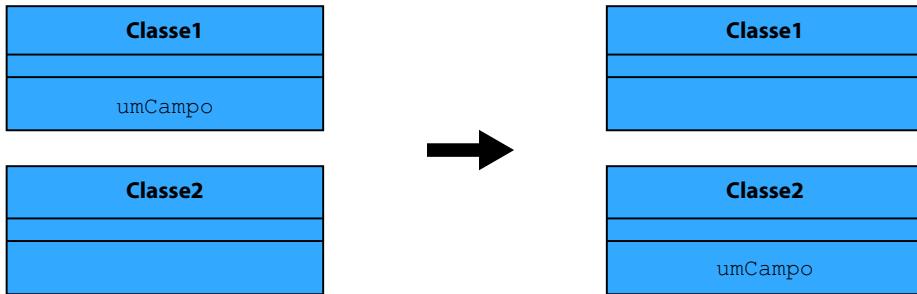


Figura 10 – Técnica Mover Campo

Fonte: adaptado de Fowler (2004).

Extrair Classe

Para Fowler (2004, p. 132), se “você tem uma classe fazendo um trabalho que deveria ser feito por duas, crie uma nova classe e mova os campos e métodos pertinentes da classe antiga para a nova”.

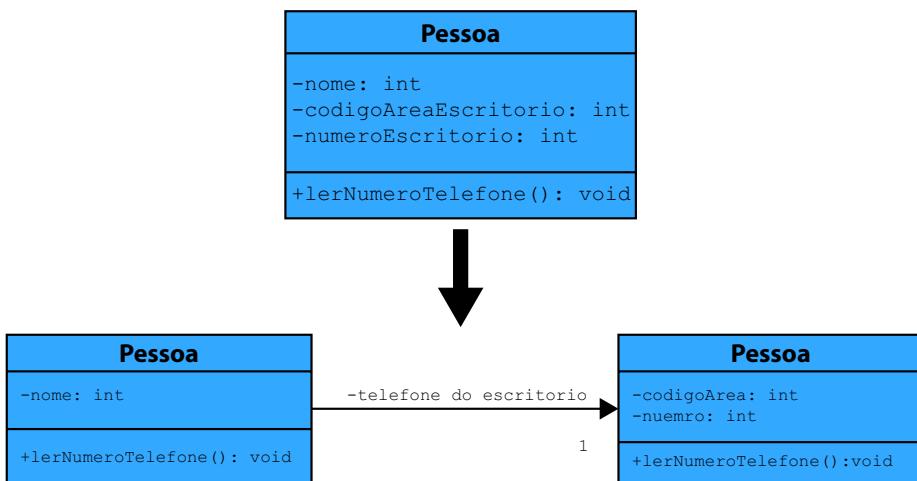


Figura 11 – Técnica Extrair Classe

Fonte: adaptado de Fowler (2004).

Simplificando Expressões Condicionais

Segundo Fowler (2004, p. 204), a lógica condicional pode ser refatorada e regra básica é “quebra de uma expressão condicional em partes. Isso é importante porque separa a lógica variante dos detalhes do que ocorre”.

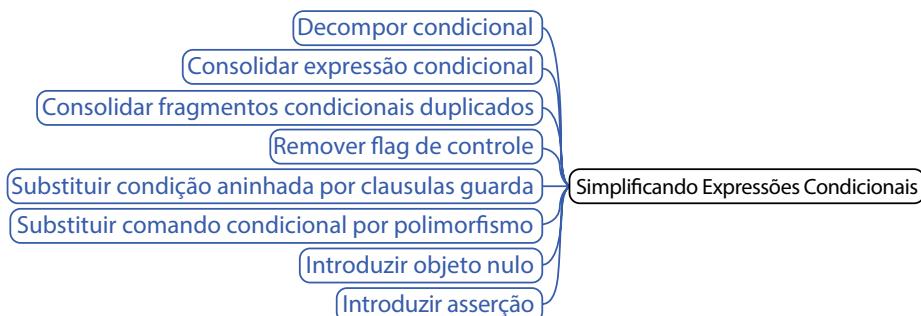


Figura 12 – Categoria Simplificando Expressões Condicionais

Fonte: adaptado de Fowler (2004).

Decompor condicional

De uma estrutura condicional complicada (*if-then-else*), extraia métodos da condição, da parte após o *then* e da parte após o *else*.

```

if (data.before (INICIO_VERAO) || data.after (FIM_VERAO)
    aCobrar = quantidade * _taxaDeInverno + _precoDoServiçoNoInverno;
else aCobrar = quantidade * _taxaDeInverno;
  
```



```

if (naoVerao(data))
    aCobrar = PrecoDeInverno (quantidade);
else aCobrar = PrecoDeVerão (quantidade);
  
```

Figura 13 – Técnica Decompor Condisional

Fonte: adaptado de Fowler (2004, p. 205).

Consolidar Expressão Condicional

De uma sequência de testes condicionais com o mesmo resultado, pode-se combiná-los em uma única expressão condicional e extraí-la.

```
double valorPorIncapacidade() {
    if (_antiguidade < 2) return 0;
    if (_mesesIncapacitado > 12) return 0;
    if (_tempoParcial) return 0;
    // calcular o valor por incapacidade
```



```
double valorPorIncapacidade() {
    if (naoPreencherRequisitosParaIncapacidade) return 0;
    // calcular o valor por incapacidade
```

Figura 14 – Técnica Consolidar Expressão Condisional

Fonte: adaptado de Fowler (2004).

Tornando as Chamadas de Métodos Mais Simples

Conforme Fowler (2004, p. 232), “este conjunto de refatorações explora a utilização da refatoração para tornar as interfaces mais diretas; produzindo interfaces que sejam mais fáceis de entender e usar”.

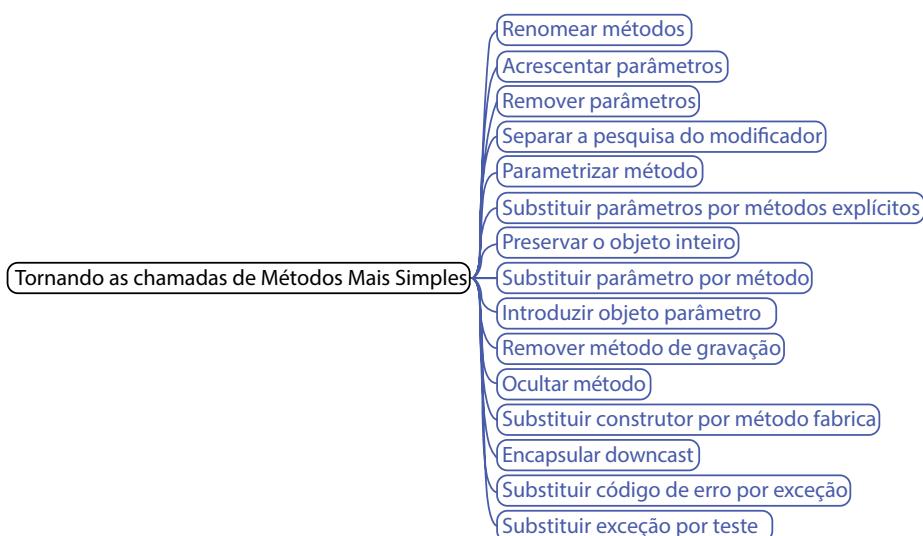


Figura 15 – Categoria Tornando as Chamadas de Métodos Mais Simples

Fonte: adaptado de Fowler (2004).

Renomear Método

Se você tem um nome de um método que não revela seu propósito, altere o nome do método.

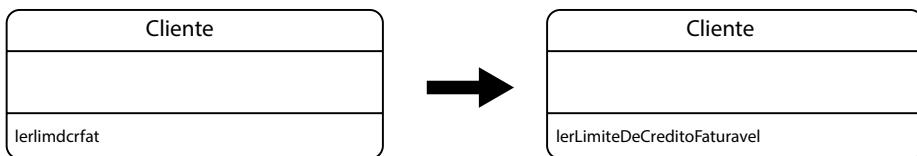


Figura 16 – Técnica Renomear Método

Fonte: adaptado de Fowler (2004).

Parametrizar Método

Se você tem métodos que fazem coisas semelhantes, mas com diferentes valores contidos no corpo do método, crie um método que use um parâmetro para estes diferentes valores.

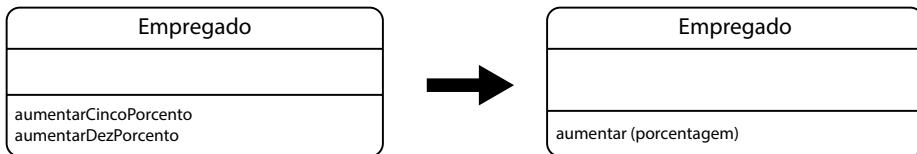


Figura 17 – Técnica Parametrizar Método

Fonte: adaptado de Fowler (2004).

Substituir Parâmetro por Método

Se você tem um objeto que chama um método e depois passa o resultado como parâmetro para um método, significa que o destinatário também pode chamar esse método, então remova o parâmetro e deixe o destinatário chamar o método.

```
int precoBase = quantia * precoItem;
nivelDesconto = lerNivelDesconto();
double precoFinal = precoComDesconto (precoBase, nivelDesconto);
```



```
int precoBase = quantia * precoItem;
double precoFinal = precoComDesconto (precoBase);
```

Figura 18 – Técnica Substituir Parâmetro por Método

Fonte: adaptado de Fowler (2004).

Organizando Dados

Segundo Fowler (2004), organizar dados é um conjunto de técnicas de refatorações que são extremamente úteis e que tornam mais fácil o tratamento de dados, utilizando linguagem de programação orientada a objetos, que podem permitir a definição de novos tipos que vão além do que pode ser feito com os tipos de dados simples das linguagens tradicionais.

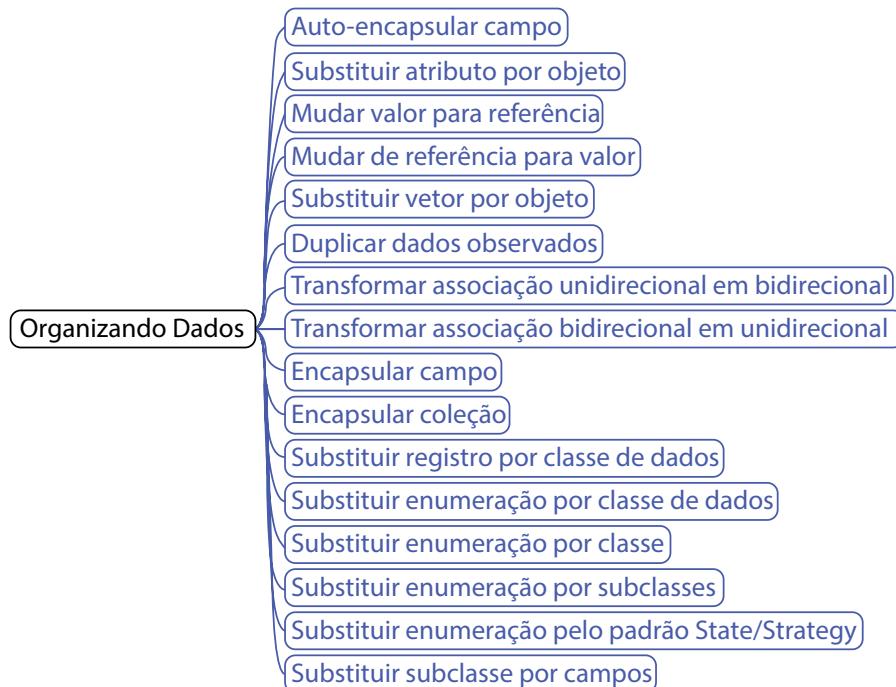


Figura 19 – Categoria Organizando Dados

Fonte: adaptado de Fowler (2004).

Auto-encapsular campo

Se você está acessando um campo diretamente, mas o acoplamento a esse campo está se tornando inadequado, então crie métodos de acesso ao campo e use apenas estes métodos para acessar o campo.

```
private int _baixo, _alto;
boolean inclui (int arg) {
    return arg >= _baixo && arg <= _alto;
```



```
private int _baixo, _alto;
boolean inclui (int arg) {
    return arg >= _lerBaixo() && arg <= _lerAlto();
}
int lerBaixo() {return _baixo;
int lerAlto() {return _alto;
```

Figura 20 – Técnica Auto-encapsular campo

Fonte: adaptado de Fowler (2004).

Substituir vetor por objeto

Se você tem um vetor no qual certos elementos significam coisas diferentes, então substitua o vetor por um objeto que tenha um campo para cada elemento do vetor.

```
String[] disputa = new String [3]
disputa[0] = "Liverpool";
disputa[1] = "15";
```



```
Desempenho disputa = new Desempenho ();
disputa.gravarNome ("Liverpool");
disputa.gravarVitorias ("15");
```

Figura 21 – Técnica Substituir vetor por objeto

Fonte: adaptado de Fowler (2004).

Encapsular Campo

Você tem um campo público, então torne-o privado e forneça métodos de acesso.

```
public String _nome;
```

```
private String _nome;  
public String lerNome() { return _nome; }  
public void gravarNome(String arg) {_nome = arg; }
```

Figura 22 – Técnica Encapsular Campo

Fonte: adaptado de Fowler (2004).

Lidando com Generalização

Conforme Fowler (2004, p. 273), “generalizações produzem seu próprio lote de refatorações, a maior parte delas lidando com a movimentação de métodos por uma hierarquia de herança”.

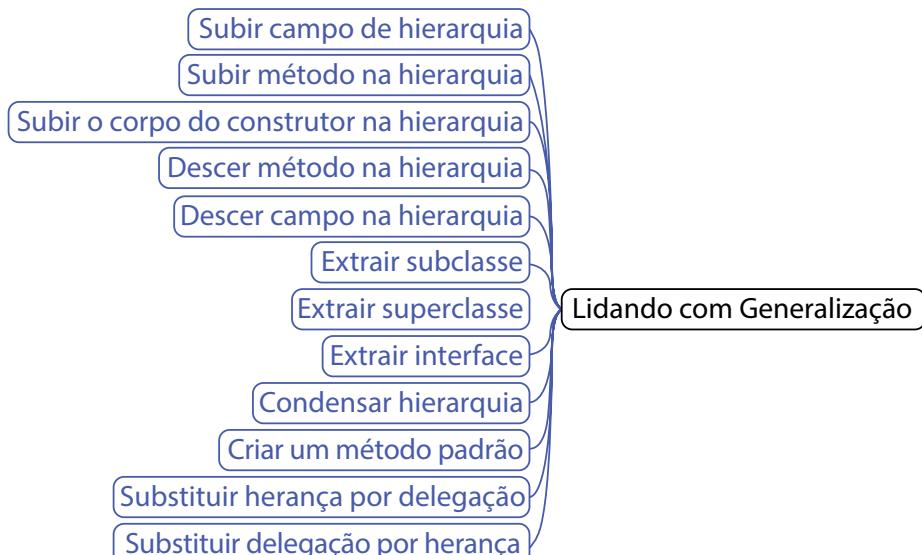


Figura 23 – Categoria Lidando com Generalização

Fonte: adaptado de Fowler (2004).

Subir Campo na Hierarquia

Você tem duas subclasses que tem o mesmo campo, então move o campo para a superclasse.

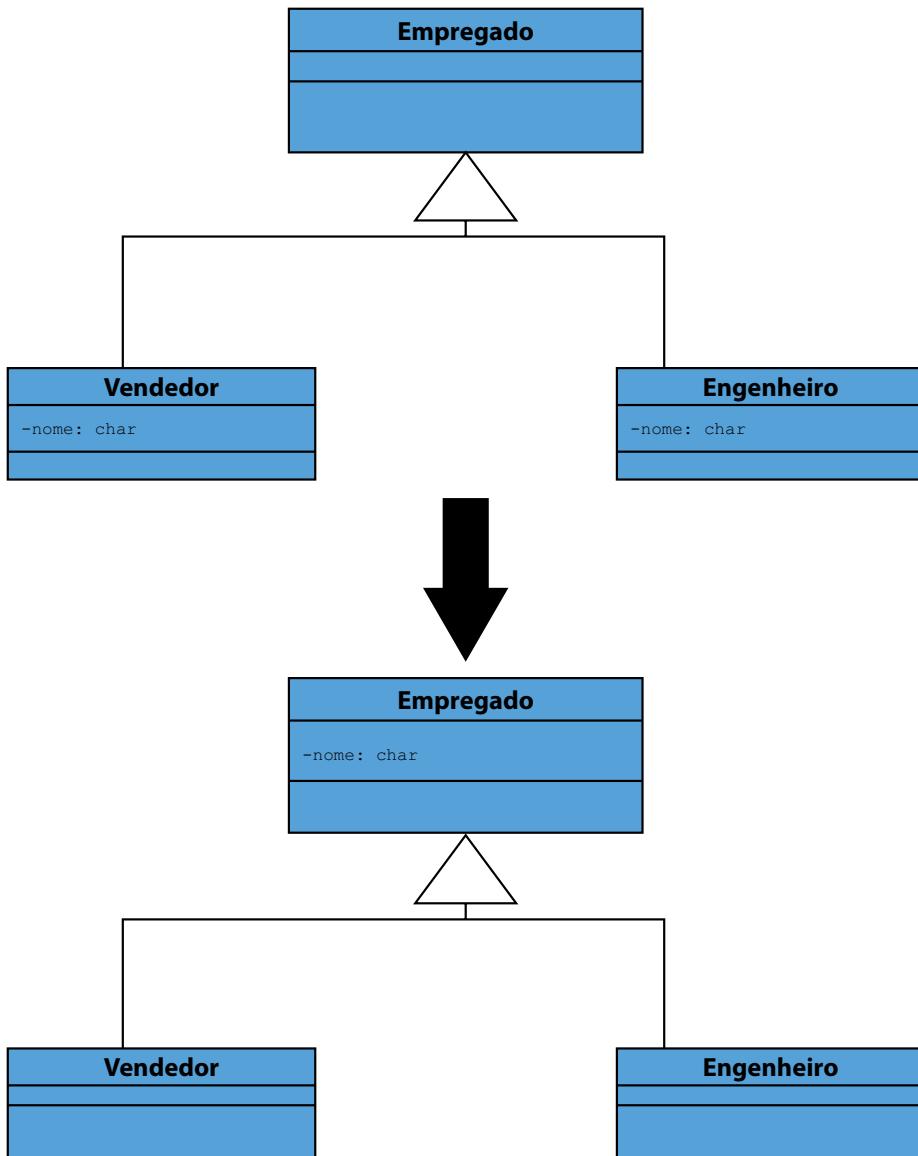


Figura 24 – Categoria Subir Campo na Hierarquia
Fonte: adaptado de Fowler (2004).

Extrair Subclasse

Se você tem uma classe com características que são usadas apenas em algumas instâncias, então crie uma subclasse para esse subconjunto de características.

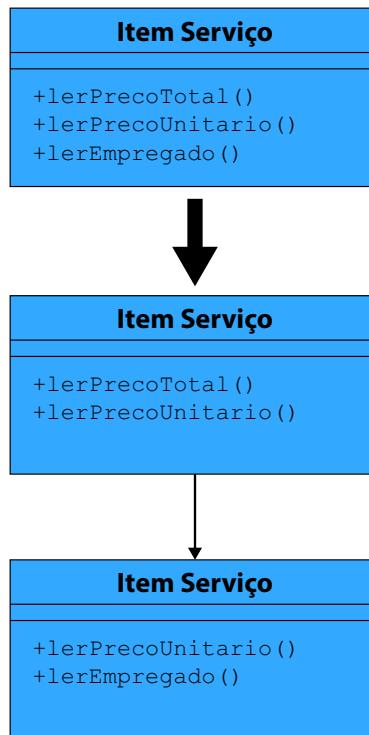


Figura 25 - Categoria Extrair Subclasse

Fonte: adaptado de Fowler (2004).

O CATÁLOGO DE PADRÕES DE PROJETO

O catálogo de padrões de projeto contém 23 padrões de projeto. Seus nomes e intenções são listados a seguir para dar uma visão geral.

Tabela 4 - Catálogo de Padrões de Projeto

Abstract Factory	Fornece uma interface para criação de famílias de objetos relacionados ou dependentes sem especificar suas classes concretas.
Adapter	Converte a interface de uma classe em outra interface esperada pelos clientes. O Adapter permite que certas classes trabalhem em conjunto, pois de outra forma seria impossível por causa de suas interfaces incompatíveis.
Bridge	Separa uma abstração da sua implementação, de modo que as duas possam variar independentemente.
Builder	Separa a construção de um objeto complexo da sua representação, de modo que o mesmo processo de construção possa criar diferentes representações.
Chain of Responsibility	Evita o acoplamento do remetente de uma solicitação ao seu destinatário, dando a mais de um objeto a chance de tratar a solicitação. Encadeia os objetos receptores e passa a solicitação ao longo da cadeia até que um objeto a trate.
Command	Encapsula uma solicitação como um objeto, desta forma permitindo que você parametrize clientes com diferentes solicitações, enfileire ou registre (<i>log</i>) solicitações e suporte operações que podem ser desfeitas.
Composite	Compõe objetos em estrutura de árvore para representar hierarquias do tipo partes-todo. O Composite permite que os clientes tratem objetos individuais e composições de objetos de maneira uniforme.
Decorator	Atribui responsabilidades adicionais a um objeto dinamicamente. Os decorators fornecem uma alternativa flexível a subclasses para extensão da funcionalidade.
Façade	Fornece uma interface unificada para um conjunto de interfaces em um subsistema. O Façade define uma interface de nível mais alto que torna o subsistema mais fácil de usar.
Factory Method	Define uma interface para criar um objeto, mas deixa as subclasses decidirem qual classe a ser instanciada. O Factory Method permite a uma classe postergar (<i>defer</i>) a instanciação às subclasses.
Flyweight	Usa compartilhamento para suportar grandes quantidades de objetos, de granularidade fina, de maneira eficiente.

Interpreter	Dada uma linguagem, define uma representação para sua gramática juntamente com um interpretador que usa a representação para interpretar sentenças nessa linguagem.
Iterator	Fornece uma maneira de acessar sequencialmente os elementos de uma agregação de objetos sem expor sua representação subjacente.
Mediator	Define um objeto que encapsula a forma como um conjunto de objetos interage. O Mediator promove o acoplamento fraco ao evitar que os objetos se refiram explicitamente uns aos outros, permitindo que você varie suas interações independentemente.
Memento	Sem violar o encapsulamento, captura e externaliza um estado interno de um objeto, de modo que possa posteriormente ser restaurado para este estado.
Observer	Define uma dependência um-para-muitos entre objetos, de modo que, quando um objeto muda de estado, todos os seus dependentes são automaticamente notificados e atualizados.
Prototype	Especifica os tipos de objetos a serem criados usando uma instância prototípica e criar novos objetos copiando esse protótipo.
Proxy	Fornece um objeto representante (<i>surrogate</i>), ou um marcador de outro objeto, para controlar o acesso a ele.
Singleton	Garante que uma classe tenha somente uma instância e fornece um ponto global de acesso para ela.
State	Permite que um objeto altere seu comportamento quando seu estado interno muda. O objeto parecerá ter mudado de classe.
Strategy	Define uma família de algoritmos, encapsula cada um deles e os torna intercambiáveis. O Strategy permite que o algoritmo varie independentemente dos clientes que o utilizam.
Template Method	Define o esqueleto de um algoritmo em uma operação, postergando a definição de alguns passos para subclasses. O Template Method permite que as subclasses redefinam certos passos de um algoritmo sem mudar sua estrutura.
Visitor	Representa uma operação a ser executada sobre os elementos da estrutura de um objeto. O Visitor permite que você defina uma nova operação sem mudar as classes dos elementos sobre os quais opera.

Fonte: adaptado de Gamma (2007, p. 24-25).

Os padrões de projeto podem variar na granularidade e no nível de abstração, e como existem muitos padrões de projeto, precisamos organizá-los de alguma forma. A organização e a classificação dos padrões de projeto ajuda a entender e como direcionar esforços na descoberta de novos. Conforme Gamma (2007) os padrões são classificados por dois critérios:

[...] o primeiro critério, chamado **finalidade**, reflete o que um padrão faz. Os padrões podem ter finalidade de criação, estrutural ou comportamental. Os padrões de criação se preocupam com o processo de criação de objetos. Os padrões estruturais lidam com a composição de classes ou de objetos. Os padrões comportamentais caracterizam as maneiras pelas quais classes ou objetos interagem e distribuem responsabilidades. O segundo critério, chamado **escopo**, especifica se o padrão se aplica primariamente a classes ou a objetos. Os padrões para classes lidam com os relacionamentos entre classes e suas subclasses. Esses relacionamentos são estabelecidos através do mecanismo de herança, assim eles são estáticos – fixados em tempo de compilação. Os padrões para objetos lidam com relacionamentos entre objetos que podem ser mudados em tempo de execução e são mais dinâmicos. Quase todos utilizam a herança em certa medida (GAMMA, 2007, p. 26).

Segundo Kerievsky (2008, p. 57) “bons projetistas refatoraram em muitas direções, sempre com o objetivo de chegar a um projeto melhor”. Muitas refatorações que são aplicadas podem envolver padrões de projetos. A direção de uma Refatoração pode ser dirigida ao uso de padrões de projeto e normalmente são influenciadas pela natureza do padrão. Por exemplo, Compor método é uma Refatoração baseada no padrão *Composed Method*.

Após a Refatoração, você deve avaliar se o seu projeto realmente melhorou. Caso não tenha melhorado, volte alguns passos ou Refatore em outra direção, tal como na direção contrária ao padrão ou use outro padrão.

CONSIDERAÇÕES FINAIS

Prezado aluno(a)! Chegamos ao final de mais uma unidade. Aprendemos conceitos e princípios sobre a refatoração para padrões e como ela pode ser aplicada de maneira controlada e eficiente, que não introduza erros no código e que melhore metodicamente a sua estrutura.

Iniciamos a unidade introduzindo algumas definições sobre a Refatoração, seus conceitos e sua história. Foi explicado que a Refatoração iniciou com Martin Fowler, em 1999, ao popularizar o termo refatoração para referir-se à atividade de aprimoramento de seu estilo de código sem alterar o seu comportamento. Juntamente com a história sobre a Refatoração, aprendemos que Fowler definiu um catálogo de sintomas, indicando que o seu código precisa de Refatoração e que ele o chamou de “catálogo de maus cheiros”.

Aprendemos também sobre os principais motivos para refatorar um código. Também falamos sobre alguns problemas que podem ocorrer com a Refatoração e que para ajudar a resolver esses problemas, é descrito como encontrar os maus cheiros no código e a limpá-los com a Refatoração.

Estudamos o catálogo de refatorações com alguns exemplos de utilização, não todos, já que explicamos que não é um catálogo definitivo, é apenas o começo. Também falamos sobre a Refatoração para padrões e como a utilização de padrões pode melhorar o seu código.

Ao longo da unidade, aprendemos como é importante manter um código limpo e como a refatoração ajuda neste processo, já que o propósito da refatoração é tornar o software mais fácil de entender e modificar. Aprendemos também que podemos fazer muitas mudanças no software que alterem pouca coisa ou nada no comportamento observável, mas que somente as mudanças feitas para tornar o software mais fácil de entender são consideradas refatorações.

Depois desta Unidade, com o conhecimento que já adquirimos podemos passar para a próxima Unidade, para que deste modo você se aprofunde ainda mais no conhecimento.

Preparados? Então vamos em frente! Bons estudos!

ATIVIDADES



1. A refatoração é considerada uma das técnicas mais poderosas para a produção de um bom código. Tsui e Karam (2013) citam o catálogo de “maus cheiros” fornecido por Fowler. Com base nestas informações, analise as sentenças.
 - I. Código duplicado mostrando desperdício, método longo, classe grande, Instruções switch.
 - II. Inveja dos funcionários e dos programadores do código.
 - III. Intimidade inapropriada, na qual uma classe refere-se a partes privadas de outras classes.
 - IV. Decisões de como o sistema irá se comportar, em termos de: estrutura de cabos, hardware, infraestrutura de telefonia, a interface.
 - V. Inveja da funcionalidade, quando um método tende a utilizar mais de um objeto de uma classe diferente a aquele que pertence.

É correto o que se afirma em:

- a) II e V apenas.
- b) I e II apenas.
- c) I, II e IV apenas.
- d) I, II e V apenas.
- e) I, II, III e IV apenas.

ATIVIDADES



2. Refatorar o código em pequenos passos ou blocos, ajuda a prevenir a introdução de defeitos. Algumas refatorações podem levar minutos para serem executadas e outras requerem um esforço sustentado por um tempo maior. Pensando sobre isso, quais as razões ou motivações mais comuns para se refatorar? Assinale a alternativa correta:
- a) Refatorar melhorar o projeto do Software, Refatorar torna o software mais fácil de projetar, Refatorar ajuda a encontrar campos e a programar mais rapidamente.
 - b) Refatorar melhora o levantamento de requisitos do Software, Refatorar torna o software mais complicado de entender, Refatorar ajuda a encontrar falhas e a programar mais corretamente.
 - c) Refatorar melhorar o projeto do Software, Refatorar torna o software mais fácil de entender, Refatorar ajuda a encontrar falhas e programar mais rapidamente.
 - d) Refatorar melhorar o projeto do Software, Refatorar torna o software mais complicado de entender, Refatorar ajuda a encontrar comentários no código e programar mais rapidamente.
 - e) Refatorar melhorar o projeto do Software, Refatorar torna o software mais fácil de entender, Refatorar ajuda a encontrar comentários e erros no código e a programar mais rapidamente.
3. Na refatoração para obter os melhores resultados, é necessário que seu código seja olhado por muitos. Segundo Kerievsky (2008) esta é uma das razões pelas quais a programação extrema sugere as práticas de programação em pares e propriedades coletivas de código. Sobre isso, porque devemos manter o código sempre limpo?

ATIVIDADES



4. O catálogo de “maus cheiros” é para ser usado quando não estiver certo sobre a refatoração a ser feita. Este catálogo ajuda a identificar o que o desenvolvedor está sentido e ajuda a identificar e detectar o mau cheiro no código e lhe apontar na direção certa, quando for realizar a refatoração. Sobre esse assunto, analise as afirmativas:

- I. Intimidade inadequada é quando às vezes as classes se tornam íntimas demais e gastam tempo demais sondando as partes privadas das outras. Classes íntimas demais precisam ser separadas.
- II. A herança pode muitas vezes levar à intimidade excessiva e as subclasses sempre saberão mais sobre seus pais do que esses gostariam que elas soubessem.
- III. Herança recusada é quando as subclasses conseguem herdar métodos e dados de seus pais, mas e se não quiserem ou precisarem do que lhes é dado? Elas recebem todos estes ótimos presentes e escolhem apenas alguns para brincar. A história tradicional é que isso significa que a hierarquia está errada.
- IV. Comentários são considerados cheiro doce, pois muitas vezes são usados como desodorantes. Pois é com muita frequência que se vê códigos cheios de comentários e percebe-se que eles estão lá porque o código é ruim.

É correto o que se apresenta em:

- a) III apenas.
- b) I e II apenas.
- c) I, II e IV apenas.
- d) I, II e III apenas.
- e) I, II, III e IV apenas.

5. O catálogo de refatorações, segundo Fowler (2004), não é de forma alguma completo ou fechado, mas deve fornecer uma base sólida para o seu próprio trabalho de refatoração. Pensando sobre isso, descreva sobre como deve ser o formato de refatorações que foi criado por Fowler.



Padrões de projeto : soluções reutilizáveis de software orientado a objetos

Projetar software orientado a objetos é difícil, mas projetar software *reutilizável* orientado a objetos é ainda mais complicado. Você deve identificar objetos pertinentes, fatorá-los em classes no nível correto de granularidade, definir as interfaces das classes, as hierarquias de herança e estabelecer as relações-chave entre eles. O seu projeto deve ser específico para o problema a resolver, mas também genérico o suficiente para atender problemas e requisitos futuros. Também deseja evitar o reprojeto, ou pelo menos minimizá-lo. Os mais experientes projetistas de software orientado a objetos lhe dirão que um projeto reutilizável e flexível é difícil, senão impossível, de obter corretamente da primeira vez. Antes que um projeto esteja terminado, eles normalmente tentam reutilizá-lo várias vezes, modificando-o a cada vez.

Projetistas experientes realizam bons projetos, ao passo que novos projetistas são sobrecarregados pelas opções disponíveis, tendendo a recair em técnicas não orientadas a objetos que já usavam antes. Leva um longo tempo para os novatos aprenderem o que é realmente um bom projeto orientado a objetos. Os projetistas experientes evidentemente sabem algo que os inexperientes não sabem. O que é? Uma coisa que os melhores projetistas sabem que não devem fazer é resolver cada problema a partir de princípios elementares ou do zero. Em vez disso, eles reutilizam soluções que funcionaram no passado. Quando encontram uma boa solução, eles a utilizam repetidamente. Consequentemente, você encontrará padrões, de classes e de comunicação entre objetos, que reaparecem frequentemente em muitos sistemas orientados a objetos. Esses padrões resolvem problemas específicos de projetos e tornam os projetos orientados a objetos mais flexíveis e, em última instância, reutilizáveis. Eles ajudam os projetistas a reutilizar projetos bem-sucedidos ao basear os novos projetos na experiência anterior. Um projetista que está familiarizado com tais padrões pode aplicá-los imediatamente a diferentes problemas de projeto, sem necessidade de redescobri-los.

Uma analogia nos ajudará a ilustrar este ponto. Os novelistas ou autores de roteiros (cinema, teatro, televisão) raramente projetam suas tramas do zero. Em vez disso, eles seguem padrões como "O herói tragicamente problemático"

(Macbeth, Hamlet, etc.) ou "A Novela Romântica" (um sem-número de novelas de romances). Do mesmo modo, projetistas de software orientado a objetos seguem padrões como "represente estados como objetos" e "adorne objetos de

maneira que possa facilmente acrescentar/remover características". Uma vez que você conhece o padrão, uma grande quantidade de decisões de projeto decorre automaticamente.

Todos sabemos o valor da experiência de projeto. Quantas vezes você já não passou pela experiência do *déjà vu* durante um projeto – aquele sentimento de que já resolveu um problema parecido antes, embora não sabendo exatamente onde e como? Se pudesse lembrar os detalhes do problema anterior e de que forma o resolveu, então poderia reutilizar a experiência em lugar de redescobri-la. Contudo, nós não fazemos um





bom trabalho ao registrar experiência em projeto de software para uso de outros. Cada padrão de projeto sistematicamente nomeia, explica e avalia um aspecto de projeto importante e recorrente em sistemas orientados a objetos. O nosso objetivo é capturar a experiência de projeto de uma forma que as pessoas possam usá-la efetivamente.

Os padrões de projeto tornam mais fácil reutilizar projetos e arquiteturas bem sucedidas. Expressar técnicas testadas e aprovadas as torna mais acessíveis para os desenvolvedores de novos sistemas. Os padrões de projeto ajudam a escolher alternativas de projeto que tornam um sistema reutilizável e a evitar alternativas que comprometem a reutilização. Os padrões de projeto podem melhorar a documentação e a manutenção de sistemas ao fornecer uma especificação explícita de interações de classes e objetos e o seu objetivo subjacente. Em suma, ajudam um projetista a obter mais rapidamente um projeto adequado.

Nenhum dos padrões de projeto descreve projetos novos ou não-testados. Incluímos somente projetos que foram aplicados mais de uma vez em diferentes sistemas. Muitos deles nunca foram documentados antes. São parte do folclore da comunidade de desempenho de software orientado a objetos ou elementos de sistemas orientados a objetos bem-sucedidos — em nenhum dos casos é fácil para projetistas novatos aprender as lições. Assim, embora esses projetos não sejam novos, nós os capturamos numa forma nova e acessível: como um catálogo de padrões, que tem um formato consistente.

Apesar do tamanho do livro, os padrões apresentados capturam somente uma fração do que um especialista pode conhecer. Não há nenhum padrão que lide com concorrência ou programação distribuída ou programação para tempo real.

Padrões de projeto não são projetos, como listas encadeadas e tabelas de acesso aleatório, que podem ser codificadas em classes e ser reutilizadas tais como estão. Tampouco são projetos complexos, de domínio específico, para uma aplicação inteira ou subsistema. Um padrão de projeto nomeia, abstrai e identifica os aspectos-chave de uma estrutura de projeto comum para torná-la útil para a criação de um projeto orientado a objetos reutilizável. O padrão de projeto identifica as classes e instâncias participantes, seus papéis, colaborações e a distribuição de responsabilidades. Cada padrão de projeto focaliza um problema ou tópico particular de projeto orientado a objetos. Ele descreve em que situação pode ser aplicado, se ele pode ser aplicado em função de outras restrições de projeto e as consequências, custos e benefícios de sua utilização.

A escolha da linguagem de programação é importante porque influencia o ponto de vista do projetista (usuário do padrão): nossos padrões assumem recursos de linguagem do nível do Smalltalk/C++, e essa escolha determina o que pode, ou não, ser implementado facilmente. Se tivéssemos assumido o uso de linguagens procedurais, deveríamos ter incluído padrões de projetos como “Herança”, “Encapsulamento” e “Polimorfismo”. Os padrões de projeto variam na sua granularidade e no seu nível de abstração.

Fonte: Gamma et al. (2007, p. 14-18).

MATERIAL COMPLEMENTAR



LIVRO

Refatoração para Padrões

Joshua Kerievsky

Editora: Bookman

Sinopse: este livro mostra a conexão entre padrões de projeto e refatoração, uma das práticas-chave de programação eXtrema (XP). Adotando padrões ao estilo do livro clássico de Gamma e cols, Padrões de Projeto, e usando código do mundo real, Kerievsky documenta o raciocínio e os passos que fazem parte de muitas das transformações de projeto baseadas em padrões. Inclui maneiras práticas de começar a trabalhar, de grande valor até para iniciantes em padrões ou refatoração.



NA WEB

O que é e o que não é refatoração – de acordo com Kent Beck e Martin Fowler. Artigo interessante sobre o que não é refatoração de acordo com Kent Beck e Martin Fowler. O texto mostra conceitos do que é refatoração e compara com o que não é de acordo com os autores. Para saber mais sobre o artigo, acesse o link:<https://www.ibm.com/developerworks/community/blogs/fd26864d-cb41-49cf-b719-d89c6b072893/entry/o_que_C3_A9_e_o_que_n_C3_A3o_C3_A9_refatora_C3_A7_C3_A3o_de_acordo_com_kent_beck_e_martin_fowler?lang=en>.



NA WEB

Introdução a Refatoração

Artigo que fala sobre os conceitos da refatoração em código de software . Mostrando que fatorar significa tornar o código mais comprehensível e bem estruturado alternando o seu design interno de modo a facilitar o desenvolvimento em equipe e a manutenção , e ai estão os grandes ganhos em refatorar o código. Para ficar por dentro do assunto do artigo, acesse: <<http://www.linhadecodigo.com.br/artigo/2832/introducao-a-refatoracao.aspx#ixzz50hAKGmgm>>.

REFERÊNCIAS

FOWLER, M. **Refatoração:** Aperfeiçoando o Projeto de Código Existente, tradução Acauan Fernandes. Porto Alegre: Bookman, 2004.

GAMMA, E. et al. **Padrões de projeto:** soluções reutilizáveis de software orientado a objetos. Porto Alegre : Bookman, 2007.

KERIEVSKY, J. **Refatoração para Padrões,** tradução Eduardo Kessler Piveta. Porto Alegre: Bookman, 2008.

TSUI, F.; KARAN, O. **Fundamentos de engenharia de software.** 2. ed. Rio de Janeiro: LTC, 2013.



GABARITO

1. Opção correta é a C.
2. Opção correta é a C.
3. Devemos manter o código sempre limpo procurando remover códigos duplicados, para deixá-los simplificados e claros. Código limpo ajuda no desenvolvimento rápido, que leva a clientes e programadores felizes e ao sucesso do sistema. Em desenvolvimento ágil, se faz necessário a refatoração para deixar o código sempre limpo, pois sem refatoração, as mudanças no código tornam-se cada vez mais difíceis e caras. A refatoração pode melhorar a qualidade do código e facilitar as mudanças no sistema. A refatoração é considerada uma das técnicas mais poderosa para a produção de um bom código.
4. Opção correta é a E.
5. O Formato de Refatorações criado por Fowler tem cinco partes. O **nome** é importante para construir um vocabulário de refatorações. Um breve **resumo** da situação atual na qual precisa da refatoração e um resumo do que ela faz. O resumo ajuda a refatorar mais rapidamente. A **motivação** que ajuda a descrever porque a refatoração deve ser feita e as circunstâncias nas quais não deve ser feita. A **mecânica** que é uma descrição concisa, passo a passo, de como executar a Refatoração e **exemplos** que mostra um uso bem simples da refatoração para ilustrar como ela funciona.



TENDÊNCIAS EMERGENTES DA ENGENHARIA DE SOFTWARE

UNIDADE

V

Objetivos de Aprendizagem

- Entender Sistemas Distribuídos.
- Compreender a Arquitetura Arquitetura Orientada a Serviço (SOA).
- Estudar as Tendências Leves em Engenharia de Software.

Plano de Estudo

A seguir, apresentam-se os tópicos que você estudará nesta unidade:

- Sistemas Distribuídos
- Arquitetura Orientada a Serviço (SOA – *Service Oriented Architecture*)
- Tendências Leves

INTRODUÇÃO

Estamos chegamos ao final do nosso livro e, nesta unidade, vamos aprender sobre as Tendências Emergentes da Engenharia de Software e como essas tendências que têm um efeito sobre a tecnologia de engenharia de software, seus cenários de negócios, organizacionais, mercado e cultural.

Vamos começar falando sobre Sistemas Distribuídos, que é um sistema que envolve vários computadores, diferente dos sistemas centralizados, em que todos os componentes do sistema executam em um único computador. Vamos aprender que os sistemas distribuídos são mais complexos e por isso difíceis de projetar, implementar e testar do que os sistemas centralizados. Também vamos compreender a imprevisibilidade inerente à operação de sistemas distribuídos, a qual deve ser considerada pelo projetista do sistema e porque ela é tão importante e vamos conhecer os cinco padrões de arquitetura de sistemas distribuídos.

Em seguida, estudaremos sobre as arquiteturas orientadas a serviços (SOA) que são uma forma de desenvolvimento de sistemas distribuídos em que os componentes de sistema são serviços autônomos, executando em computadores geograficamente distribuídos. Vamos aprender que os serviços são plataforma e implementação independentes de linguagem e da aplicação que o usa e que a engenharia de serviços é o processo de desenvolvimento de serviços para reúso em aplicações orientadas a serviços.

E por fim, falaremos sobre as tendências leves e como as novas tecnologias são introduzidas regularmente, apresentadas como sendo a “solução” para muitos dos problemas que os engenheiros de software enfrentam e incorporadas nos projetos grandes e pequenos. Vamos aprender que quando uma nova tecnologia é introduzida, ela passa por um ciclo de vida que nem sempre leva a uma aceitação ampla, apesar de as expectativas originais serem elevadas. Qualquer tecnologia de engenharia de software ganha uma aceitação ampla que está ligado a sua habilidade para resolver os problemas apresentados pelas tendências tanto leves (*soft*) quanto pesadas (*hard*).

Preparado(a) para começar? Então, vamos seguir em frente. Boa leitura e bons estudos.



SISTEMAS DISTRIBUÍDOS

Segundo Sommerville (2011), a grande maioria dos grandes sistemas computacionais são sistemas distribuídos. Segundo o autor, um sistema distribuído é um sistema que envolve vários computadores, diferente dos sistemas centralizados, em que todos os componentes do sistema executam em um único computador. Tanenbaum e Van Steen definem um sistema distribuído como sendo:

[...] um sistema distribuído é um conjunto de computadores independentes que se apresenta a seus usuários como um sistema único e coerente. Essa definição tem vários aspectos importantes. O primeiro é que um sistema distribuído consiste em componentes (isto é, computadores) autônomos. Um segundo aspecto é que os usuários, sejam pessoas ou programas, acham que estão tratando com um único sistema. Isto significa que, de um modo ou de outro, os componentes autônomos precisam colaborar. Como estabelecer essa colaboração é o cerne do desenvolvimento de sistemas distribuídos (TANENBAUM; VAN STEEN, 2007, p. 1).

A definição de sistema distribuído para Couloris et al (2013, p. 01) “um sistema distribuído é aquele no qual os componentes localizados em computadores interligados em rede se comunicam e coordenam suas ações apenas passando mensagens”.

Essa definição nos leva a pensar de forma abrangente, onde todos os sistemas com computadores interligados em rede podem ser distribuídos de maneira útil, e se eles estão conectados por meio de uma rede, podem estar separados por qualquer distância, podendo estar em continentes separados, no mesmo edifício ou na mesma sala (COULOURIS et al, 2013).

E conforme Coulouris et al (2013), podemos identificar as seguintes vantagens ao utilizar a abordagem distribuída de desenvolvimento de sistemas: (tabela 1).

Tabela 1 - Vantagens ao utilizar a abordagem distribuída

Compartilhamento de recursos	Um sistema distribuído permite o compartilhamento de recursos de hardware e software (impressoras), recursos de dados (como arquivos) e recursos com funcionalidade mais específica (como os mecanismos de busca).
Sistemas Abertos	Um sistema computacional é aberto quando ele pode ser estendido e reimplementado de várias maneiras. O fato de um sistema distribuído ser ou não um sistema aberto é determinado principalmente pelo grau com que novos serviços de compartilhamento de recursos podem ser adicionados e disponibilizados para uso por uma variedade de programas clientes.
Concorrência	Tanto os serviços como os aplicativos fornecem recursos que podem ser compartilhados pelos clientes em um sistema distribuído. Portanto, existe a possibilidade de que vários clientes tentem acessar um recurso compartilhado ao mesmo tempo.
Escalabilidade	Os sistemas distribuídos funcionam de forma efetiva e eficaz em muitas escalas diferentes, variando desde uma pequena intranet até a Internet. Um sistema é descrito como <i>escalável</i> se permanece eficiente quando há um aumento significativo no número de recursos e no número de usuários.
Tratamento de Falhas	Às vezes, os sistemas de computador falham. Quando ocorrem falhas no <i>hardware</i> ou no <i>software</i> , os programas podem produzir resultados incorretos ou podem parar antes de terem concluído a computação pretendida. As falhas em um sistema distribuído são parciais – isto é, alguns componentes falham, enquanto outros continuam funcionando. Portanto, o tratamento de falhas é particularmente difícil.

Fonte: adaptado de Coulouris et al. (2013, p. 16-25).

O desejo de compartilhar recursos, segundo Coulouris et al (2013) é uma das principais motivação para se construir e usar sistemas distribuídos.

O termo “recurso” é bastante abstrato, mas caracteriza bem o conjunto de coisas que podem ser compartilhadas de maneira útil em um sistema de computadores interligados em rede. Ele abrange desde componentes de *hardware*, como discos e impressoras, até entidades definidas pelo *software*, como arquivos, bancos de dados e objetos de dados de todos os tipos. Isso inclui o fluxo de quadros de vídeo proveniente de uma câmera de vídeo digital ou a conexão de áudio que uma chamada de telefone móvel representa (COULOURIS et al, 2013, p. 3).

Os sistemas distribuídos são mais complexos e por isso difíceis de projetar, implementar e testar do que os sistemas centralizados. Para Sommerville (2011, p. 334), “é mais difícil compreender as propriedades emergentes de sistemas distribuídos por causa da complexidade das interações entre os componentes do sistema e sua infraestrutura”. O sistema distribuído depende da largura da banda de rede, da carga de rede e da velocidade de todos os computadores que fazem parte do sistema em vez de o desempenho do sistema depender da velocidade de execução de um processador. E com isso, mover os recursos de uma parte do sistema para outra pode afetar drasticamente o desempenho do sistema (SOMMERVILLE, 2011).



QUESTÕES SOBRE SISTEMAS DISTRIBUÍDOS

Como vimos no tópico anterior, os sistemas distribuídos são mais complexos do que os executados em um único processador. Segundo Sommerville (2011, p. 334), “esta complexidade surge porque é praticamente impossível ter um modelo de controle top-down desses sistemas”, pois muitas vezes, o sistema que fornece funcionalidades são sistemas independentes sem nenhuma autoridade sobre eles, já que a rede que os conecta é um sistema gerenciado separadamente.

O sistema é complexo e que não pode ser controlado pelos proprietários dos sistemas que usam a rede. Para Sommerville (2011, p. 334), “portanto, essa é uma imprevisibilidade inerente à operação de sistemas distribuídos, a qual deve ser considerada pelo projetista do sistema”.

Mas como lidar com esta imprevisibilidade? Devemos considerar nos sistemas distribuídos algumas questões mais importantes de projeto, que são:

Tabela 2 - Questões importantes de projeto

1. Transparência	Em que medida o sistema distribuído deve aparecer para o usuário como um único sistema? Quando é útil aos usuários entender que o sistema é distribuído?
2. Abertura	Um sistema deveria ser projetado usando protocolos-padrão que ofereçam suporte à interoperabilidade ou devem ser usados protocolos mais especializados que restrinjam a liberdade do projetista?
3. Escalabilidade	Como o sistema pode ser construído para que seja escalável? Ou seja, como todo o sistema poderia ser projetado para que sua capacidade possa ser aumentada em resposta às crescentes exigências feitas ao sistema?
4. Proteção	Como podem ser definidas e implementadas as políticas de proteção que se aplicam a um conjunto de sistemas gerenciados independentemente?
5. Qualidade de serviço	Como a qualidade do serviço que é entregue aos usuários do sistema deve ser especificada e como o sistema deve ser implementado para oferecer uma qualidade aceitável e serviço para todos os usuários?
6. Gerenciamento de falhas	Como as falhas do sistema podem ser detectadas, contidas (para que elas tenham efeitos mínimos em outros componentes do sistema) e reparadas?

Fonte: adaptado de Sommerville (2011, p. 335).

Um sistema distribuído não é transparente para os usuários, mas um sistema transparente significa que os usuários veriam o sistema como um único sistema cujo comportamento não é afetado pela maneira como o sistema é distribuído. E para Sommerville (2011) isso na prática, é impossível de se alcançar.

O controle central de um sistema distribuído é impossível, e, como resultado, os computadores individuais em um sistema podem ter comportamentos diferentes em momentos diferentes. Além disso, como sempre leva um tempo determinado para os sinais viajarem através de uma rede, os atrasos na rede são inevitáveis. O comprimento desses atrasos depende da localização dos recursos no sistema, da qualidade da conexão da rede do usuário e da carga de rede. A abordagem de projeto para atingir a transparência depende de se criarem abstrações dos recursos em um sistema distribuído de modo que a realização física desses recursos possa ser alterada sem a necessidade de alterações no sistema de aplicação. Na prática, é impossível fazer um sistema completamente transparente e, geralmente, os usuários estão conscientes de que estão lidando com um sistema distribuído. (SOMMERVILLE, p. 335).

Temos os **sistemas distribuídos abertos** que são sistemas construídos de acordo com normas que são geralmente aceitas pela comunidade. Neste caso, os componentes de qualquer fornecedor podem ser integrados ao sistema e podem interoperar com outros componentes do sistema. Os componentes de sistema devem ser desenvolvidos independentemente em qualquer linguagem de programação e, se estas estiverem em conformidade com as normas, funcionarão com outros componentes (SOMMERVILLE, 2011).

MODELOS DE INTERAÇÃO

Em um sistema distribuído existem dois tipos fundamentais de interação que podem ocorrer: interação procedural e interação baseada em mensagens.

- **Interação procedural:** envolve um computador que chama um serviço conhecido oferecido por algum outro computador e (normalmente) esperando que esse serviço seja fornecido.

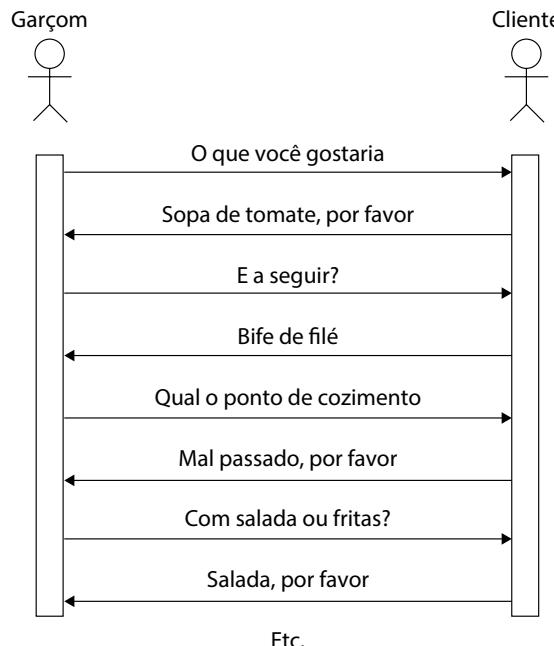


Figura 1 - Interação procedural entre um cliente e um garçom

Fonte: Sommerville (2011, p. 337).

- **Interação baseada em mensagens:** envolve o computador “que envia” que define as informações sobre o que é requerido em uma mensagem, que são enviadas para outro computador.

```

<entrada>
  <nome do prato = "sopa" type = "tomate"/>
  <nome do prato = "sopa" type = "peixe"/>
  <nome do prato = "salada de pombo"/>
</entrada>
<curso principal>
  <nome do prato = "bife" type = "lombo" cozinar = "médio"/>
  <nome do prato = "bife" type = "filé" cozinar = "mal passado"/>
  <nome do prato = "robalo"/>
</principal>
<acompanhamento>
  <nome do prato = "batatas fritas" porções = "2"/>
  <nome do prato = "salada" porções = "1"/>
</acompanhamento>
  
```

Figura 2 - Interação baseada em mensagens entre um garçom e o pessoal da cozinha

Fonte: Sommerville (2011, p. 337).

Nas figuras 1 e 2, foi ilustrada a diferença entre a interação procedural e a interação baseada em mensagens, onde consideramos a situação no qual o usuário está pedindo uma refeição em um restaurante. Quando o usuário tem uma conversa com o garçom, ele está envolvido em uma série de interações síncronas, procedurais que define qual vai ser o seu pedido. O usuário faz um pedido, o garçom reconhece o pedido; o usuário faz outra solicitação, a qual é reconhecida, e assim por diante. Para Sommerville (2011, p. 337), “isso é comparável aos componentes interagindo em um sistema de software em que um componente chama métodos de outros componentes”.

Ao encerrar o pedido do usuário, o garçom passa esse pedido para a cozinha preparar a refeição. Nesse momento, isso é a interação baseada em mensagens, que mostra o processo síncrono de pedido como uma série de chamadas. O garçom pega o pedido dos usuários como uma série de interações, onde cada interação é parte do pedido, mas com a cozinha, o garçom tem uma única interação, onde a mensagem define o pedido completo.

MIDDLEWARE

Conforme Sommerville (2011, p. 338) “os componentes em um sistema distribuído podem ser implementados em diferentes linguagens de programação e podem ser executados em diferentes tipos de processador”. Assim como os modelos de dados, as representações de informações, os protocolos usados para comunicação podem ser todos diferentes.

Um sistema distribuído requer um software que possa gerenciar essas diversas partes e assegurar que elas podem se comunicar e trocar dados. O termo ‘middleware’ é usado para se referir a esse software — ele fica no meio, entre os componentes distribuídos do sistema. Normalmente, o middleware é implementado como um conjunto de bibliotecas que é instalado em cada computador distribuído, além de um sistema de run-time para gerenciar a comunicação. O middleware é um software de uso geral geralmente comprado no mercado e que não é escrito especialmente por desenvolvedores de aplicações (SOMMERVILLE, 2011, p. 338).

Exemplos de middleware:

- Software para Gerenciamento de Comunicações com Bancos de Dados.
- Gerenciadores de Transações.
- Conversores de Dados.
- Controladores de Comunicação.

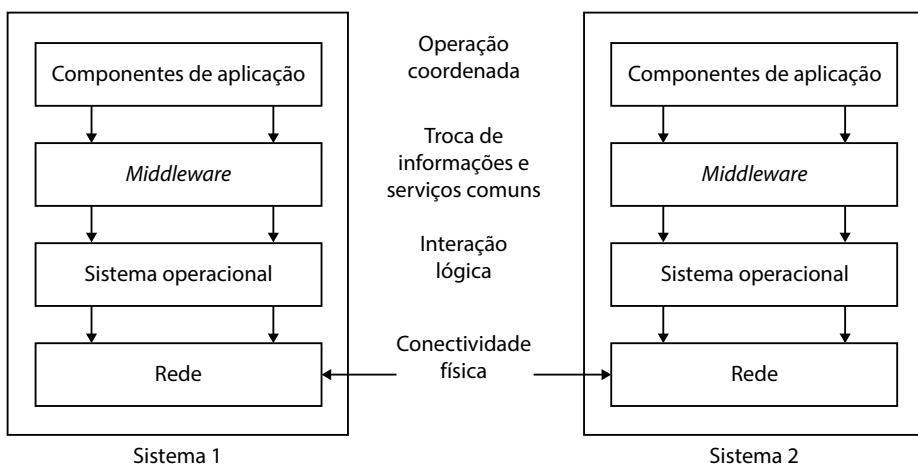


Figura 3 - Middleware em um sistema distribuído

Fonte: Sommerville (2011, p. 338).

O middleware costuma fornecer dois tipos distintos de suporte em um sistema distribuído:

Tabela 3 - Tipos distintos de suporte fornecido pelo *Middleware*

1. Suporte a interações	O middleware coordena as interações entre diferentes componentes do sistema. O middleware fornece transparência da localização, assim não é necessário que os componentes saibam os locais físicos dos outros componentes. Ele também pode suportar a conversão de parâmetros se diferentes linguagens de programação forem usadas para implementar componentes, detecção de eventos e comunicação etc.
2. A prestação de serviços comuns	O middleware fornece implementações reusáveis de serviços que podem ser exigidas por vários componentes do sistema distribuído. Usando esses serviços comuns, os componentes podem, facilmente, interoperar e prestar serviços de usuário de maneira consistente.

Fonte: adaptado de Sommerville (2011, p. 339).

PADRÕES DE ARQUITETURA PARA SISTEMAS DISTRIBUÍDOS

Para Sommerville (2011, p. 341) “os projetistas de sistemas distribuídos precisam organizar seus projetos de sistema para encontrar um equilíbrio entre desempenho, confiança, proteção e capacidade de gerenciamento do sistema”. Como não existe um modelo universal de organização de sistema distribuído surgiram vários padrões de arquitetura. Ao projetar um sistema distribuído, deve-se escolher um padrão de arquitetura que ofereça suporte aos requisitos não funcionais críticos de seu sistema.

Segundo Sommerville (2011), tem cinco padrões de arquitetura de sistemas distribuídos:

- 1. Arquitetura de mestre-escravo:** são comumente usadas em sistemas de tempo real em que tempos de resposta precisos de interação são requeridos e quando é importante cumprir os deadlines (prazos) de processamento. Esse modelo de mestre-escravo de um sistema distribuído pode ser usado em situações em que seja possível ou necessário prever o processamento distribuído.

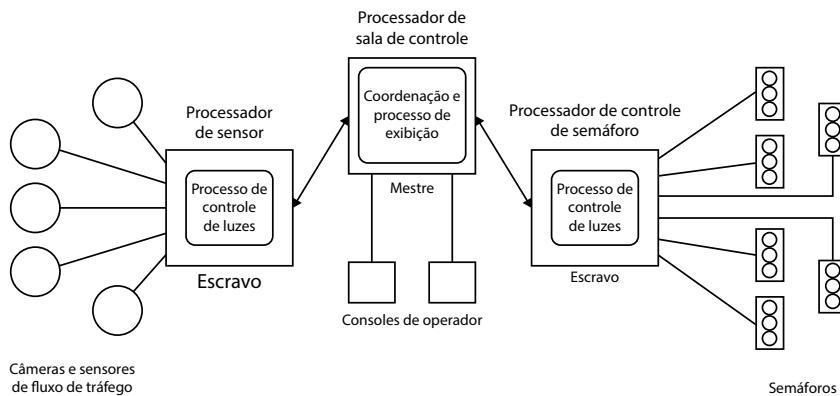


Figura 4 - Um sistema de gerenciamento de tráfego com uma arquitetura mestre-escravo
Fonte: Sommerville (2011, p. 342).

2. Arquitetura cliente-servidor de duas camadas: usada para sistemas cliente-servidor simples e em situações nas quais é importante centralizar o sistema por razões de proteção. Sistemas cliente-servidor tem a parte do sistema de aplicação que é executada no computador do usuário (o cliente) e tem a parte que é executada em um computador remoto (o servidor). Uma arquitetura cliente-servidor de duas camadas é a forma mais simples da arquitetura cliente-servidor. O sistema é implementado como um único servidor lógico e, também, um número indefinido de clientes que usam esse servidor (SOMMERVILLE, 2011).

A figura 5 mostra duas formas desse modelo de arquitetura:

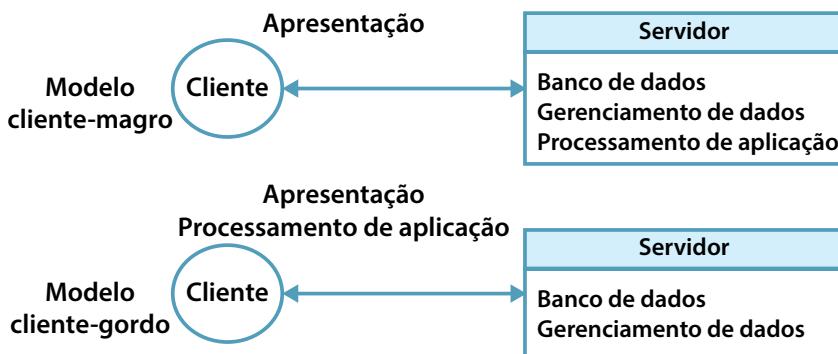


Figura 5 - Modelos de arquitetura cliente-magro e cliente-gordo

Fonte: Sommerville (2011, p. 342).

Tabela 4 - Formas do Modelo de arquitetura cliente-servidor de duas camadas

Modelo cliente-magro	A camada de apresentação é implementada no cliente e todas as outras camadas (gerenciamento de dados, processamento de aplicação e banco de dados) são implementadas em um servidor.
Modelo cliente-gordo	Em que parte ou todo o processamento de aplicação é executado no cliente e as funções de banco de dados e gerenciamento são implementadas no servidor.

Fonte: adaptado de Sommerville (2011, p. 342).

3. Arquitetura cliente-servidor multicamadas: usada quando existe um alto volume de transações a serem processadas pelo servidor. As diferentes camadas do sistema, apresentação, gerenciamento de dados, processamento de aplicação e banco de dados, são processos separados que podem ser executados em diferentes processadores (SOMMERVILLE, 2011).

**Camada 1.
Apresentação**

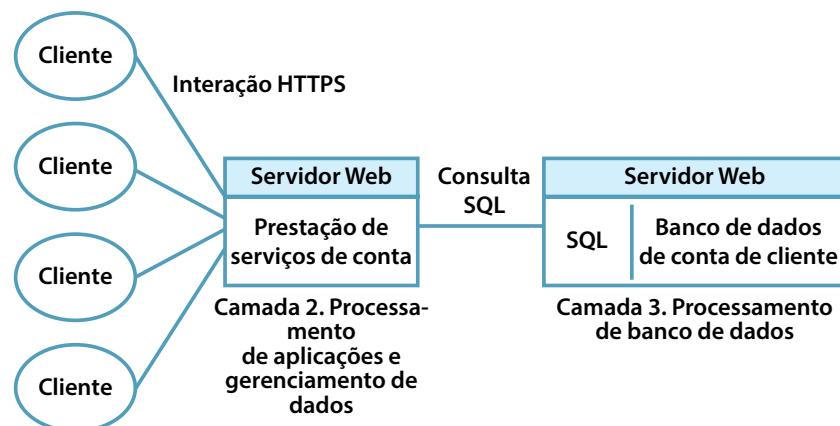


Figura 6 - Arquitetura de três camadas para um sistema de Internet Banking
Fonte: Sommerville (2011, p. 344).

4. Arquitetura distribuída de componentes: usada quando recursos de diferentes sistemas e bancos de dados precisam ser combinados ou é usada como um modelo de implementação para sistemas cliente-servidor em várias camadas.

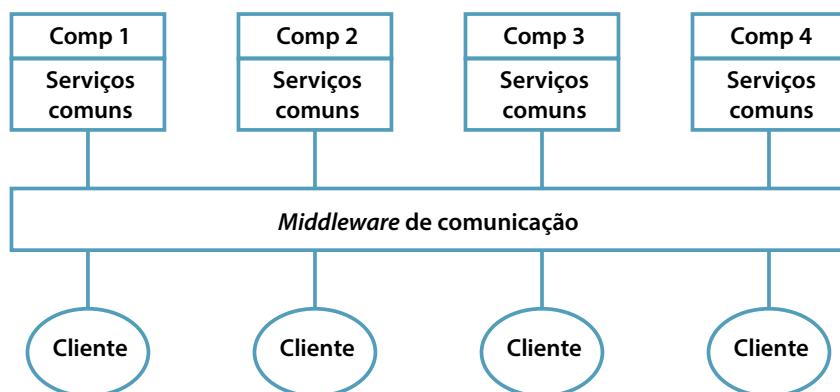


Figura 7 - Arquitetura distribuída de componentes
Fonte: Sommerville (2011, p. 345).

Projetar o sistema como um conjunto de serviços, sem tentar alocar esses serviços nas camadas do sistema, é a abordagem mais geral de projetar sistemas distribuídos. Onde cada serviço, ou grupo de serviços relacionados, é implementado usando um componente separado (SOMMERVILLE, 2001).

5. Arquitetura ponto-a-ponto: usada quando os clientes trocam informações localmente armazenadas e o papel do servidor é introduzir clientes uns aos outros.

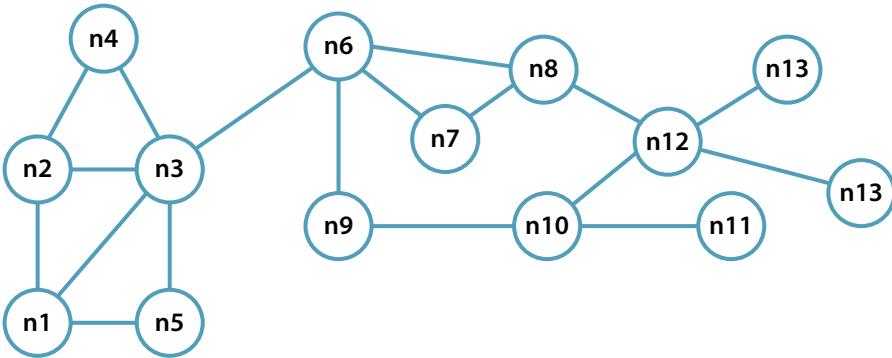


Figura 8 - Arquitetura p2p descentralizada

Fonte: Sommerville (2011, p. 347).

Segundo Sommerville (2011, p. 347) “os sistemas ponto-a-ponto (p2p, do inglês peer-to-peer) são sistemas descentralizados em que os processamentos podem ser realizados por qualquer nó na rede”. Em sistemas ponto a ponto, todo o sistema é projetado para aproveitar o poder computacional e o armazenamento disponível por meio de uma rede potencialmente enorme de computadores.

Em um sistema computacional p2p em que uma computação de processador intensivo é distribuída por meio de um grande número de nós, é normal que alguns nós sejam superpontos. Seu papel é distribuir o trabalho para outros nós, conferir e verificar os resultados da computação. As arquiteturas ponto-a-ponto permitem o uso eficiente da capacidade por meio de uma rede. No entanto, os principais problemas que têm inibido seu uso são as questões de proteção e confiança. A comunicação ponto-a-ponto envolve abrir seu computador para direcionar as interações com outros pontos, e isso significa que esses sistemas poderiam, potencialmente, acessar qualquer um de seus recursos (SOMMERVILLE, 2011, p. 349).



SAIBA MAIS

Os benefícios de sistemas distribuídos são aqueles que podem ser dimensionados para lidar com o aumento da demanda, podem continuar a fornecer serviços de usuário (mesmo que algumas partes do sistema falhem) e habilitam o compartilhamento de recursos. Os servidores fornecem gerenciamento de dados, de aplicações e de banco de dados. Os sistemas cliente-servidor podem ter várias camadas, com diferentes camadas do sistema distribuídas em computadores diferentes. Os padrões de arquitetura para sistemas distribuídos incluem arquiteturas mestre-escravo, arquiteturas cliente-servidor de duas camadas e de múltiplas camadas, arquiteturas de componentes distribuídos e arquiteturas ponto-a-ponto. Os componentes de sistemas distribuídos requerem middleware para lidar com as comunicações de componentes e para permitir que componentes sejam adicionados e removidos do sistema. As arquiteturas ponto a ponto são arquiteturas descentralizadas em que não existe distinção entre clientes e servidores.

ARQUITETURA ORIENTADA A SERVIÇO (SOA – SERVICE ORIENTED ARCHITECTURE)

Conforme Sommerville (2011, p. 356), “as arquiteturas orientadas a serviços (SOA) são uma forma de desenvolvimento de sistemas distribuídos em que os componentes de sistema são serviços autônomos, executando em computadores geograficamente distribuídos”.

Os serviços são plataforma e implementação independentes de linguagem e da aplicação que o usa.

Os sistemas de software podem ser construídos pela composição de serviços locais e serviços externos de provedores diferentes, com interação perfeita entre os serviços no sistema. Os provedores de serviços podem desenvolver serviços especializados e oferecerem para uma variedade de usuários de serviço de diferentes organizações. Os provedores de serviços projetam e implementam serviços e especificam a interface para esses serviços. Eles também publicam informações sobre esses serviços em um registro acessível. Os solicitantes (às vezes chamados clientes de serviço) de serviço que desejam usar um serviço descobrem a especificação desse serviço e localizam o provedor de serviço. Em seguida, eles podem ligar sua aplicação a esse serviço específico e comunicar-se com ele, usando protocolos de serviço-padrão (SOMMERVILLE, 2011, p. 356).



Figura 9 - Arquitetura orientada a serviço

Fonte: Sommerville (2011, p. 356).

Desde o surgimento do SOA, houve um processo de padronização. Muitas empresas de hardware e software estão comprometidas com esses padrões e como resultado, a SOA não sofreu as incompatibilidades que costumam surgir com as inovações técnicas, em que diferentes fornecedores mantêm sua versão proprietária da tecnologia (SOMMERVILLE, 2011).

A figura 10 mostra os padrões fundamentais criadas para oferecer suporte aos web services.

Os protocolos de web services cobrem todos os aspectos das SOA, desde os mecanismos básicos para troca de informações de serviço (SOAP) até os padrões de linguagem da programação (WS-BPEL). Esses padrões são todos baseados em XML, uma notação legível por máquina e humanos que permite a definição de dados estruturados, em que o texto é marcado com um identificador significativo. XML tem uma gama de tecnologias, como XSD para definição de esquemas, que são usadas para estender e manipular descrições de XML (SOMMERVILLE, 2011, p. 356).

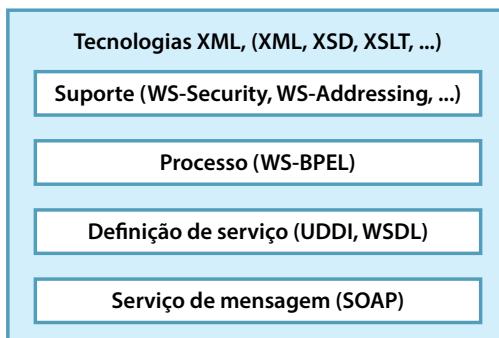


Figura 10 - Padrões de web services

Fonte: Sommerville (2011, p. 356).

Os principais padrões para SOA de Web são:

Tabela 5 - Principais Padrões para SOA de Web

SOAP	Esse é um padrão de trocas de mensagens que oferece suporte à comunicação entre os serviços. Ele define os componentes essenciais e opcionais das mensagens passadas entre serviços.
WSDL	A linguagem de definição de web service (do inglês Web Service Definition Language) é um padrão para a definição de interface de serviço. Define como as operações de serviço (nomes de operação, parâmetros e seus tipos) e associações de serviço devem ser definidas.
WS-BPEL	Esse é um padrão para uma linguagem de workflow, que é usada para definir programas de processo que envolvem vários serviços diferentes.

Fonte: adaptado de Sommerville (2011, p. 357).

A arquitetura SOA são menos rígidas, onde as ligações de serviços podem mudar durante a sua execução. E para Sommerville (2011, p. 357), “isso significa que uma versão diferente, mas equivalente, do serviço, pode ser executada em diferentes momentos”. Muitos sistemas são construídos exclusivamente para o uso de web services, e outros podem misturar os web services com componentes desenvolvidos localmente.

Como exemplo de como as aplicações que usam uma mistura de componentes e serviços podem ser organizadas (figura 11) considere o seguinte cenário proposto por Sommerville:

[...] um sistema de informações em um carro fornece aos motoristas informações sobre clima, condições de tráfego da estrada, informações locais, e assim por diante. Ele é ligado ao rádio do carro para que a informação seja entregue como um sinal em um canal de rádio específico. O carro é equipado com receptores GPS para descobrir sua posição, e, com base nessa posição, o sistema acessa uma gama de serviços de informação. Em seguida, as informações podem ser entregues na linguagem especificada pelo motorista (SOMMERVILLE, p. 357).

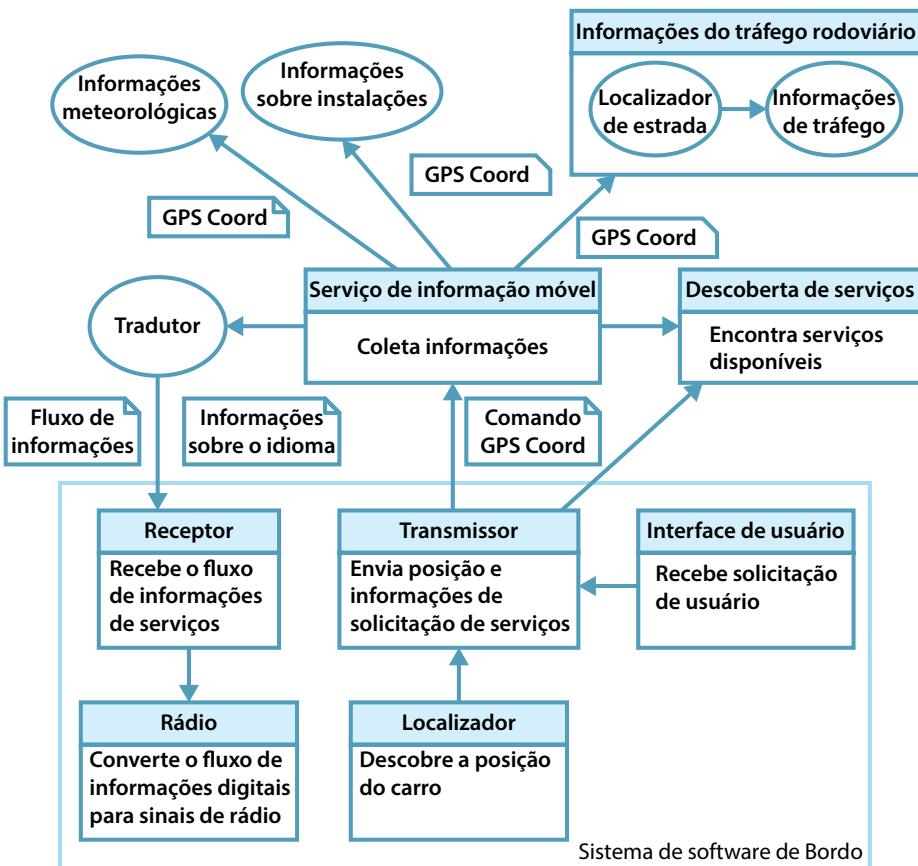


Figura 11 - Um sistema de informações de bordo baseado em serviços

Fonte: Sommerville (2011, p. 358).

O exemplo descrito acima mostra uma das principais vantagens da abordagem orientada a serviços, que conforme Sommerville (2011, p. 358) “não é necessário decidir quando o sistema é programado ou implantado, qual provedor de serviço deve ser usado ou quais serviços específicos devem ser acessados”. Pois conforme o carro se move, o sistema de informações de bordo usa o serviço de descoberta para encontrar o serviço de informações mais adequado e os vincula. Ainda podemos mover-se além das fronteiras usando um serviço de tradução, disponibilizando as informações locais para as pessoas que não falam a língua local (SOMMERVILLE, 2011).



SAIBA MAIS

O software de bordo inclui cinco módulos, os quais lidam com comunicações com o motorista, com um receptor GPS que relata a posição do carro e com o rádio do carro. Os módulos Transmissor e Receptor lidam com todas as comunicações com os serviços externos. O carro comunica-se com um serviço móvel externo de informação que agrupa informações de uma variedade de outros serviços, fornecendo informações sobre clima, tráfego e instalações locais. Provedores diferentes em diferentes lugares oferecem esses serviços, e o sistema de bordo usa um serviço de descoberta para localizar serviços de informações adequadas e se ligarem a eles. O serviço de descoberta também é usado pelo serviço de informação móvel para conectar os serviços adequados sobre clima, tráfego e recursos. Serviços trocam mensagens SOAP que incluem informações sobre a posição do GPS usada pelos serviços para selecionar as informações apropriadas. A informação agrupada é enviada para o carro por meio de um serviço que converte essas informações na linguagem preferida do motorista.

Fonte: Sommerville (2011, p. 358).

SERVIÇOS COMO COMPONENTES REUSÁVEIS

Segundo Sommerville (2011, p. 359), “um serviço pode ser definido como um componente de software de baixo acoplamento, reusável, que encapsula funcionalidade discreta, que pode ser distribuída e acessada por meio de programas”. O modelo de componentes é um conjunto de padrões associados com web services, pois os serviços são um desenvolvimento natural dos componentes de software. Para Sommerville (2011, p. 359) a diferença entre um serviço e um componente:

[...] Uma distinção fundamental entre um serviço e um componente de software, conforme definido na CBSE, é que os serviços devem ser independentes e fracamente acoplados; ou seja, eles sempre devem operar da mesma maneira, independentemente de seu ambiente de execução. Sua interface é uma interface ‘provides’ que permite o acesso à funcionalidade de serviço. Os serviços são projetados para serem independentes e podem ser usados em contextos diferentes. Portanto, eles não têm uma interface ‘requires’ que, em CBSE, define os outros componentes do sistema que devem estar presentes.

Ainda sobre a distinção fundamental entre um serviço e um componente de software, temos que os serviços se comunicam por meio de troca de mensagens, expressas em XML, e essas mensagens são distribuídas usando protocolos-padrão de transporte de Internet (HTTP e TCP/IP). E para Sommerville (2011, p. 359) “um serviço define o que precisa de outro serviço, definindo seus requisitos em uma mensagem e enviando-a a esse serviço”. O serviço de recepção analisa a mensagem, com isso ele efetua o processamento, e envia uma resposta para os serviços solicitantes, como uma mensagem. O serviço passa a analisar a resposta para extrair as informações que são necessárias. Diferente dos componentes de software, os serviços não fazem uso de chamadas de procedimentos ou de métodos remotos para acessar a funcionalidade associada a outros serviços (SOMMERVILLE, 2011).

Quando você pretende usar um web service, precisa saber onde se encontra o serviço (sua URI) e os detalhes de sua interface. Estes são descritos em uma descrição de serviço expressa em uma linguagem baseada em XML, chamada WSDL. A especificação WSDL define três aspectos de um web service: o que faz o serviço, como ele se comunica e onde o encontrar:

Tabela 6 - Aspectos de um web service

O que	O que de um documento WSDL, denominado interface, especifica quais operações o serviço suporta e define o formato das mensagens que são enviadas e recebidas pelo serviço.
O como	O como de um documento WSDL, denominado ligação, mapeia a interface abstrata para um conjunto concreto de protocolos. A ligação especifica os detalhes técnicos de como se comunicar com um web service.
Onde	Onde de um documento WSDL descreve o local da implementação de um web service específico (seu ponto final).

Fonte: adaptado de Sommerville (2011, p. 359).

A seguir, a figura 12, mostra o modelo conceitual WSDL com os elementos de uma descrição de serviço, onde cada um é expresso em XML e pode ser fornecido em arquivos separados.

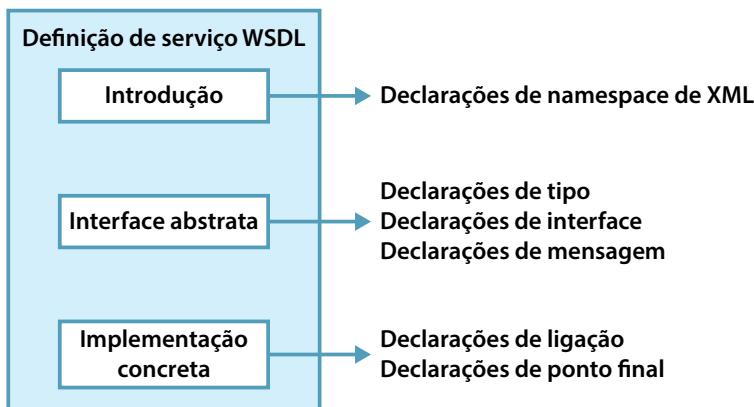


Figura 12 - Modelo Conceitual da organização de uma especificação WSDL

Fonte: Sommerville (2011, p. 360).

As partes da definição de serviço WSDL são:

Tabela 7 - Definição de serviço WSDL

Introdução	Uma parte introdutória que geralmente define os <i>namespaces</i> de XML usados e que pode incluir uma seção de documentação, fornecendo informações adicionais sobre o serviço.
Declaração de Tipo	Uma descrição opcional dos tipos usados em mensagens trocadas pelo serviço.
Declaração de Interface	Uma descrição da interface de serviço; ou seja, as operações que fornecem o serviço para outros serviços ou usuários.
Declaração de Mensagem	Uma descrição das mensagens de entrada e saída processadas pelo serviço.
Declaração de Ligação	Uma descrição da ligação usada pelo serviço (ou seja, o protocolo de mensagens que será usado para enviar, receber mensagens). O default é SOAP, mas outras ligações também podem ser especificadas. A ligação define como as mensagens de entrada e saída associadas ao serviço devem ser empacotadas em uma mensagem e especifica os protocolos de comunicação usados. A ligação também pode especificar como são incluídas as informações de apoio, como as credenciais de proteção ou identificadores de transação.
Declaração de Ponto Final	Como um identificador de recurso uniforme (URI, do inglês Uniform Resource Identifier) — o endereço de um recurso que pode ser acessado pela Internet.

Fonte: adaptado de Sommerville (2011, p. 360).

A definição de interface de serviço não inclui quaisquer informações sobre a semântica do serviço ou suas características não funcionais, como desempenho e confiança e isso é considerado um grande problema com WSDL, pois ele é apenas uma simples descrição da assinatura de serviço, ou seja, somente as operações e seus parâmetros (SOMMERVILLE, 2011, p. 360).

Ao planejar usar o serviço, é necessário que se defina o que o serviço realmente vai fazer e o que significa cada um dos diferentes campos nas mensagens de entrada e saída. E por meio de experimentação, o desempenho e a confiança precisam ser descobertos. A documentação e os nomes significativos são importantes para que os leitores consigam compreender a funcionalidade oferecida do serviço.

ENGENHARIA DE SERVIÇOS

Conforme Sommerville (2011, p. 361), “a engenharia de serviços é o processo de desenvolvimento de serviços para reúso em aplicações orientadas a serviços”. Nela, os engenheiros de serviço precisam garantir que o serviço que vai ser oferecido represente uma abstração reusável que poderia ser usada em outros sistemas. A funcionalidade a ser projetada e desenvolvida deve ser útil e que assegure que o serviço seja robusto e confiável. O serviço deve ser documentado para que possa ser usado por outros usuários em potencial.

Temos três estágios lógicos no processo de engenharia de serviço:

Tabela 8 - Estágios Lógicos no Processo de Engenharia de Serviço

Identificação de serviço candidato	Identificação dos possíveis serviços que podem ser implementados e definição dos requisitos de serviço.
Projeto de serviço	Fase em que é planejado a lógica e as interfaces de serviço WSDL.
Implementação e implantação de serviço	Fase em que é implementado e testado os serviços tornando-os disponíveis para uso.

Fonte: adaptado de Sommerville (2011, p. 361).

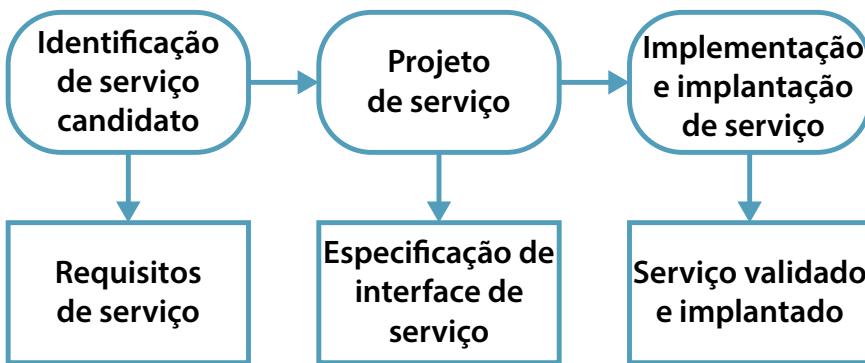


Figura 13 - Processo de Engenharia de Serviços

Fonte: Sommerville (2011, p. 361).

Quando desenvolvemos um componente reusável, podemos começar com um componente já existente e que já foi implementado e usado em outra aplicação. Com o serviço é o mesmo - podemos começar o processo com um serviço existente ou um componente que será convertido em um serviço.

Os serviços devem apoiar os processos de negócios, pois as empresas possuem uma gama de processos com muitos serviços possíveis que podem ser implementados. Para isso, é necessária a identificação de serviço candidato. A identificação de um serviço candidato envolve a compreensão e a análise dos processos de negócios da empresa. Após a análise, passa-se a decidir quais serviços reusáveis podem ser implementados para dar suporte a esses processos da empresa.

Após a seleção dos serviços candidatos, o próximo passo do processo de engenharia de serviço é projetar as interfaces de serviço. E para Sommerville (2011, p. 361) isso envolve:

[...] a definição das operações associadas com o serviço e seus parâmetros. Você também deve pensar cuidadosamente sobre o projeto de operações e mensagens de serviço. Seu objetivo deve ser minimizar o número de trocas de mensagens que deve acontecer para se completar a solicitação de serviço. É necessário garantir que tanta informação quanto possível seja passada para o serviço por uma mensagem, em vez de interações síncronas de serviço. Você também deve se lembrar de que os serviços não têm estado, e gerenciar estado específico da aplicação do serviço é de responsabilidade do usuário do serviço, e não do serviço em si. Você pode, por conseguinte, ter de passar essas informações de estado para e dos serviços em mensagens de entrada e saída.

Depois de ter selecionado os serviços candidatos e projetado suas interfaces, a fase final do processo de engenharia de serviço é a implementação de serviço. A implementação de serviço pode envolver programação usando uma linguagem de programação padrão, como Java ou C#, pois elas incluem bibliotecas com amplo suporte para o desenvolvimento de serviços.



SAIBA MAIS

O desenvolvimento de software usando serviços baseia-se na ideia de compor e configurar serviços para criar novos serviços compostos. Estes podem ser integrados com uma interface de usuário implementada em um browser para criar uma aplicação Web ou podem ser usados como componentes em alguma outra composição de serviço. Os serviços envolvidos na composição podem ser especialmente desenvolvidos para a aplicação, podem ser desenvolvidos dentro de uma empresa de serviços de negócios ou podem ser serviços de um fornecedor externo. Atualmente, muitas empresas estão convertendo suas aplicações corporativas em sistemas orientados a serviços, em que o bloco básico de construção de aplicações é um serviço, e não um componente. Isso abre a possibilidade de reúso mais generalizado dentro da empresa. A realização final da visão de longo prazo de SOA contará com o desenvolvimento de um 'mercado de serviços', em que os serviços serão comprados de fornecedores externos.

Fonte: Sommerville (2011, p. 363).

TENDÊNCIAS LEVES

Ao longo da história relativamente recente da engenharia de software, muitos pesquisadores com a ajuda de profissionais desenvolveram vários modelos de processo, métodos e ferramentas automatizadas para apoiar a mudança na maneira como desenvolvemos software. Entretanto será que encontramos a “solução mágica” de desenvolver software grandes e complexos, sem confusão, enganos e atrasos? Todavia pela história, no entanto, ainda estamos a procura da solução mágica.

Para Pressman (2016, p. 839), “novas tecnologias são introduzidas regularmente, apresentadas como sendo a “solução” para muitos dos problemas que os engenheiros de software enfrentam e incorporadas nos projetos grandes e pequenos”. Quando surgem novas tecnologias, críticos do setor exageram na importância dessas “novas” tecnologias de software, e com isso os especialistas de software as adotam com entusiasmo, mas muitas vezes tendem a não produzir o resultado esperado, e, como consequência, a busca contínua (PRESSMAN, 2016).

Quais seriam as “grandes questões” quando consideramos a evolução da tecnologia? Na tabela a seguir, alguns desafios que enfrentamos ao tentarmos isolar tendências significativas em novas tecnologias.

Tabela 9 - Desafios e tendências significativas em novas tecnologias

Que fatores determinam o sucesso de uma tendência?	O que caracteriza as tendências bem-sucedidas: Seu mérito técnico? Sua habilidade para abrir novos mercados? Sua habilidade para alterar a economia dos mercados existentes?
Qual é o ciclo de vida de uma tendência?	Embora a visão tradicional de que as tendências evoluem ao longo de um ciclo de vida bem definido e previsível, que vai da pesquisa a um produto acabado, por meio de um processo de transferência, descobrimos que muitas encerraram esse ciclo ou seguiram outro.
Com que antecedência pode uma tendência bem-sucedida ser identificada?	Se soubermos identificar fatores de sucesso e/ou se entendermos o ciclo de vida de uma tendência, podemos detectar sinais precoces de êxito. Retoricamente, buscamos a habilidade para reconhecer a próxima tendência antes dos outros.
Quais os aspectos controláveis da evolução?	Podem as corporações usar sua influência no mercado para impor tendências? Pode o governo usar seus recursos para impor tendências? Qual é o papel das normas e padrões na definição das tendências? Uma cuidadosa análise comparativa entre Ada e Java, por exemplo, poderia ser esclarecedora nesse aspecto?

Fonte: adaptado de Pressman (2016, p. 839).

Para estas questões não há uma resposta simples, pois não há dúvida de que as tentativas para identificar tecnologias significativas vão continuar evoluindo. Em muitos livros da área, se têm discutido tecnologias emergentes e seu impacto sobre a engenharia de software. Muitas foram amplamente adotadas pela comunidade e outras nunca alcançaram seu potencial. As tecnologias vêm e vão e com isso,

podemos observar que o progresso na engenharia de software será orientado pelas tendências nos negócios, organizações, mercado e tendências culturais. São estas questões que levam à inovação tecnológica.

Neste tópico da última unidade do nosso livro, vamos examinar algumas tendências tecnológicas na engenharia de software, com ênfase sobre algumas tendências nos negócios, nas organizações, no mercado e culturais que podem ter influência importante sobre a tecnologia de engenharia de software.

EVOLUÇÃO DA TECNOLOGIA

Já pensou em quanto rápido uma tecnologia evolui? Pressman (2016, p. 840) afirma que “quando uma tecnologia bem-sucedida é introduzida, o conceito inicial transforma-se em um ciclo de vida da inovação razoavelmente previsível”.

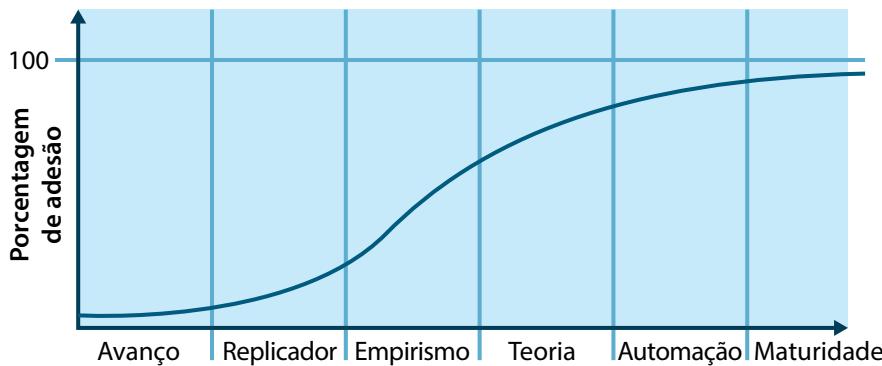


Figura 14 - Ciclo de vida da evolução tecnológica

Fonte: Pressman (2016, p. 840).

O ciclo de vida da evolução da tecnologia possui as seguintes fases:

- **Avanço:** fase onde um problema é identificado e tentativas repetidas são realizadas em busca de uma solução viável. Em algum ponto, aparece uma promessa de solução.
- **Replicador:** fase onde o trabalho inicial de avançar é reproduzido e ganha um uso mais amplo.
- **Empirismo:** fase em que se leva à criação de regras que regem o uso da tecnologia.

- **Teoria:** fase onde o sucesso repetido leva a uma **teoria** de uso mais amplo.
- **Automação:** fase onde são criadas as ferramentas de automatização.
- **Maturidade:** fase onde a tecnologia amadurece e passa a ser amplamente utilizada.

Muitas dessas tendências tecnológicas nunca atingem a maturidade. Na visão de Pressman (2016, p. 841) “a grande maioria das tecnologias “promissoras” no domínio da engenharia de software suscita um grande interesse por alguns anos e depois passa a ser usada por um grupo dedicado de usuários”. Entretanto isso não significa que algumas não tenham valor, mas que o caminho para o sucesso por meio da inovação é longo e difícil.

A evolução tecnológica irá acelerar a um passo cada vez mais rápido, chegando finalmente à era de inteligência não biológica que se combinará com a inteligência humana e a ampliará de maneira fascinante. E tudo isso, não importa como evolua, necessitará de software e sistemas que na comparação fazem nossos esforços atuais parecerem infantis. Por volta de 2040, uma combinação de computação extrema, nanotecnologia, redes com larguras de banda extremamente altas e robótica nos levará a um mundo diferente. O software, possivelmente na forma em que ainda não podemos compreender, continuará existindo no centro desse mundo novo. A engenharia de software não irá acabar (PRES-SMAN, 2016, p. 841).



REFLITA

O “ciclo da excelência” apresenta uma visão realística da integração da tecnologia no curto prazo. No entanto, a tendência no longo prazo é exponencial.

(Pressman)

Ao longo de suas carreiras, os engenheiros de software enfrentam vários tipos de desafios, como lidar com rápidas mudanças, incerteza e emergência, confiabilidade, diversidade e interdependência. Entretanto eles também passam por muitas oportunidades de fazer contribuições significativas a comunidade.

As tendências nas pesquisas “são motivadas por percepções gerais do estado da arte e da prática, por percepções do pesquisador sobre as necessidades dos profissionais, por programas de fundos nacionais que buscam estratégias específicas e por puro interesse técnico”. Tendências tecnológicas ocorrem quando pesquisas são extrapoladas para atender às necessidades da indústria e demandas do marketing (PRESSMAN, 2016, p. 842).

Todo país com empresas de TI possuem um considerável conjunto de características que definem como os negócios são conduzidos, como as dinâmicas nas empresas surgem, seus aspectos de marketing e a sua cultura dominante e a interação humana. Mas algumas tendências em cada área se tornaram universais.

Temos a conectividade e colaboração que permitem a existência de equipes de software que não ocupam o mesmo espaço físico, onde uma equipe colabora com outras que estão separadas por fuso horário, língua e cultura. Com isso a globalização leva à força de trabalho diversificada (em termos de linguagem, cultura, solução de problemas, filosofia de administração, prioridades de comunicação e interação entre as pessoas) e isso exige uma estrutura organizacional flexível (PRESSMAN, 2016).



SAIBA MAIS

Considere as interfaces para um sistema de 1 bilhão de linhas de código, para o mundo exterior, para outros sistemas interoperáveis, para a Internet (ou seu sucessor), e para os milhões de componentes internos que devem funcionar todos juntos para fazer esse monstro da computação operar corretamente. Há uma maneira confiável de garantir que todas essas conexões permitam que as informações fluam adequadamente? Considere o próprio projeto. Como administrarmos o fluxo do trabalho e acompanhamos o progresso? As abordagens convencionais poderão ser escaladas muitas vezes em ordem de grandeza? Considere o número de pessoas (e suas localizações) que estarão fazendo o trabalho, a coordenação dos profissionais e da tecnologia, o fluxo ininterrupto de alterações, a possibilidade de ambiente de sistema multiplataforma, multioperacional. Há uma maneira de administrar e coordenar indivíduos que estão trabalhando em um projeto enorme?

Fonte: Pressman (2016, p. 843).

GESTÃO DA COMPLEXIDADE

Quando pensamos na complexidade que envolve o software, surgem algumas questões:

- Será que há uma maneira confiável de garantir que todas as conexões permitam que a informação flua adequadamente?
- Será que conseguimos gerenciar o fluxo de trabalho e ainda acompanhar o progresso?
- Será que há uma maneira de gerenciar e coordenar indivíduos que estão trabalhando em um projeto grande?
- Será que tem como analisar dezenas de milhares de requisitos, limitações e restrições de uma forma que garanta que todas as inconsistências e ambiguidades, omissões e erros sejam imediatamente encontrados e corrigidos?
- Será que tem como criar uma arquitetura de projeto que seja robusta o bastante para lidar com o sistema desse tamanho?
- Será que tem como os engenheiros de software estabelecer um sistema de controle de alterações que irá manipular centenas de milhares de alterações?
- Será que podemos realizar verificações e validações de modo significativo? Como testamos um sistema de 1 bilhão de linhas de código?
- O que acontecerá no futuro? Será que as abordagens atuais estarão à altura das tarefas a serem realizadas?

Muitas dessas questões, somente no futuro poderão ser respondidas.

SOFTWARE ABERTO

Software aberto, segundo Pressman (2016, p. 846), “abrange inteligência ambiente, aplicações sensíveis ao contexto e computação generalizada”. Ou seja, software que é projeto para se adaptar a um ambiente que esteja em contínua mudança “auto-organizando sua estrutura e auto-adaptando seu comportamento”.

REQUISITOS EMERGENTES

Em todo início de um projeto de software, há um levantamento de requisitos e uma análise com todos os envolvidos. Em muitos casos, os clientes raramente definem requisitos “estáveis” no início do projeto. E isto indica que os engenheiros de software não podem prever onde ocorrerão as ambiguidades e inconsistências do projeto. Um fato comum a todos os projetos de software: os requisitos mudam. Pressman (2016, p. 846) afirma que:

[...] na medida em que os sistemas se tornam mais complexos, mesmo uma tentativa rudimentar de definir requisitos abrangentes está destinada ao fracasso. Uma definição de objetivos globais pode ser possível, pode ser conseguido um esboço dos objetivos intermediários, mas requisitos estáveis — sem chance! Os requisitos surgirão conforme todos os envolvidos na engenharia e construção de um sistema complexo aprenderem mais sobre esse sistema, sobre o ambiente no qual ele reside e sobre os usuários que vão interagir. Essa realidade implica uma série de tendências de engenharia de software.

Tudo começa por desenhar os modelos de processo (tradicional ou ágeis). Depois, seguimos com os métodos que resultam em modelos elaborados de requisitos, mas que poderão mudar à medida que for adquirido mais conhecimento sobre o sistema. E por último, ferramentas que suportem tanto os processos quanto os métodos e que devem facilitar a adaptação e as mudanças.

Outro aspecto dos requisitos emergentes, é que para os softwares desenvolvido até hoje, a fronteira entre o sistema baseado em software e seu ambiente externo é estável. Todavia para muitos da comunidade, esta fronteira pode mudar. E esta mudança deve ocorrer de uma maneira controlada, que permita que o software seja adaptado como parte de um ciclo de manutenção. Pelo que discutimos até aqui, essa opinião está começando a mudar, iniciando pelo software aberto que exige que os sistemas se adaptem e reajam às mudanças do ambiente, mesmo que sejam imprevistas.

Os requisitos emergentes levam à mudança, pela sua natureza. O autor Pressman (2016, p. 846) questiona: “Como controlamos a evolução de um aplicativo ou sistema amplamente usado durante toda a sua vida útil, e que efeito isso tem sobre a maneira como projetamos software?”. E isso nos leva ao fato que

conforme o número de alterações aumenta, a possibilidade de efeitos colaterais não desejados também aumenta. Quando os sistemas complexos com requisitos emergentes se tornarem comum, isso será um grande motivo de preocupação. E para isso, deve ser desenvolvido métodos e técnicas que ajudem a prever o impacto das mudanças neste sistema e com isso, moderando os efeitos colaterais indesejados.

MIX DE TALENTOS

E conforme o software fica mais complexo, a natureza de uma equipe de engenharia de software muda. As equipes se tornaram globais e a comunicações e colaboração e os requisitos emergentes (com o fluxo de mudanças resultantes) se tornarem a norma. Segundo Pressman (2016, p. 848), “cada equipe de software deve contribuir com talento criativo e habilidades técnicas para sua parte de um sistema complexo, e o processo todo deve permitir que o resultado dessas ilhas de talento se combine efetivamente”.

E o seguinte mix de talento é proposto para uma equipe de software dos sonhos:

Tabela 10 - Mix de talentos

Cérebro	É o arquiteto principal capaz de lidar com as demandas dos interessados e mapeá-las em uma estrutura de tecnologia extensível e implementável.
Data Grrl	É o guru de base de dados e estrutura de dados que “ataca vigorosamente através de linhas e colunas com profundo conhecimento da lógica de predicados e teoria dos conjuntos no que se refere ao modelo relacional”.
Blocker	É um líder técnico (gerente) que permite que a equipe trabalhe livre de interferências enquanto ao mesmo tempo garante que a colaboração está ocorrendo.
Hacker	É um programador perfeito que está à vontade com padrões e linguagens e pode usar ambas eficazmente.
Gatherer	Descobre habilmente requisitos de sistema com visão antropológica e os expressa com clareza.

Fonte: adaptado de Pressman (2016, p. 847).

Blocos básicos de software

Uma filosofia de engenharia de software é a necessidade de reutilização de código-fonte, de classes orientadas a objeto, de componentes e de padrões. Apesar de muitas empresas terem feito progresso na tentativa de capturar conhecimento e reutilizar soluções aprovadas, ainda temos várias outras que continuam a criar o software desde o início, pois acreditam no desejo de “soluções únicas”. Além da reutilização de software, há uma tendência em adotar soluções de plataforma de software reutilizáveis também, que incorporam coleções de funcionalidades relacionadas, fornecidas em uma estrutura de software integrada.

SAIBA MAIS



Mudança na percepção de “valor”

Durante os últimos 25 anos do século 20, a pergunta operativa que os homens de negócios faziam ao discutirem software era: “Por que custa tão caro?”. Essa pergunta raramente é feita hoje, e foi substituída por outra: “Por que não podemos obter esse (software e/ou produto baseado em software) mais rapidamente?”. Considerando o software para computador, nota-se que a percepção moderna está mudando do valor nos negócios (custo e lucratividade) para os valores de clientes que incluem: agilidade na entrega, riqueza de funcionalidade e qualidade geral do produto.

Fonte: Pressman (2016, p. 848).

Rumos da tecnologia

Uma tendência inegável, segundo Pressman (2016, p. 851), “os sistemas baseados em software sem dúvida se tornarão maiores e mais complexos com o passar do tempo”. E criar novas abordagens para lidar com esses sistemas é sem dúvida o grande desafio para a engenharia de software.



SAIBA MAIS

As tendências que têm um efeito sobre a tecnologia de engenharia de software muitas vezes originam-se de cenários de negócios, organizacionais, mercado e cultural. O grau segundo o qual qualquer tecnologia de engenharia de software ganha uma aceitação ampla está ligado a sua habilidade para resolver os problemas apresentados pelas tendências tanto leves (*soft*) quanto pesadas (*hard*). Tendências leves — a necessidade cada vez maior de conectividade e colaboração, projetos globais, transferência de conhecimento, o impacto das economias emergentes e a influência da própria cultura humana levam a uma série de desafios que abrangem desde o gerenciamento de complexidade e requisitos emergentes até a manipulação de um *mix* de talentos sempre em mudanças entre equipes de software geograficamente dispersas. Tendências pesadas — o ritmo sempre acelerado da mudança da tecnologia — surgem do âmbito das tendências leves e afetam a estrutura do software e o escopo dos processos e a maneira pela qual uma estrutura de processo é caracterizada.

Fonte: Pressman (2016, p. 852).

A seguir algumas características de sistemas que devem ser tratadas e consideradas pelos analistas e projetistas de engenharia de software para futuras aplicações:

Tabela 11 - Algumas características para futuras aplicações

Multifuncionalidade	À medida que os dispositivos digitais evoluíram começaram a apresentar um amplo conjunto de funções às vezes não relacionadas. Os engenheiros devem descrever o contexto detalhado no qual as funções serão fornecidas e, mais importante, devem identificar as interações potencialmente perigosas entre diferentes características do sistema.
Reatividade e temporização (<i>timeliness</i>)	Os dispositivos digitais interagem cada vez mais com o mundo real e devem reagir aos estímulos externos no tempo. Eles devem estabelecer interface com um amplo conjunto de sensores e devem responder em um intervalo de tempo apropriado para a tarefa em questão.
Novos modos de interação de usuário	As tendências abertas para software significam que novos modos de interação devem ser modelados e implementados. Independentemente do fato de essas novas abordagens usarem interfaces multitoque, reconhecimento de voz ou interfaces mentais diretas, as novas gerações de software para dispositivos digitais devem modelar as novas interfaces homem-computador.

Arquiteturas complexas	Sistemas ainda mais complexos estão no horizonte próximo, apresentando desafios significativos para os projetistas de software.
Sistemas heterogêneos distribuídos	Os componentes de tempo real de qualquer sistema embutido moderno podem ser conectados por meio de um barramento interno, uma rede sem fio ou da Internet (ou as três coisas).
Criticidade	O software tornou-se o componente pivô em todos os sistemas críticos nos negócios e em muitos sistemas em termos de segurança. Contudo, a comunidade de engenharia de software apenas começou a aplicar os princípios mais básicos de segurança de software.
Variabilidade de manutenção	A vida do software em um dispositivo digital raramente dura além de 3 a 5 anos, mas os sistemas complexos de aviação instalados em uma aeronave têm uma vida útil de pelo menos 20 anos. O software dos automóveis fica a meio termo. Isso deverá ter um impacto sobre o projeto.

Fonte: adaptado de Pressman (2016, p. 851).

Para que essas e outras características do software possam ser gerenciadas, os analistas e projetistas devem desenvolver uma filosofia de engenharia de software distribuída e colaborativa mais eficaz, melhores abordagens de requisitos de engenharia, uma abordagem mais robusta do desenvolvimento motivado por modelo e melhores ferramentas de software (PRESSMAN, 2016).



CONSIDERAÇÕES FINAIS

Prezado(a) aluno(a)! Chegamos ao final da última unidade do nosso livro. Aprendemos sobre as Tendências Emergentes da Engenharia de Software, como essas tendências têm um efeito sobre a tecnologia de engenharia de software, seus cenários de negócios, organizacionais, mercado e cultural.

Esperamos que a partir do conhecimento adquirido nas unidades, você procure pesquisar mais, estudar mais, além do que foi visto no livro, pois sempre surgem novas estratégias, novos conceitos para uma melhoria contínua do trabalho.

Iniciamos a leitura da unidade falando sobre Sistemas Distribuídos, que é um sistema que envolve vários computadores, diferente dos sistemas centralizados, em que todos os componentes do sistema executam em um único computador.

Aprendemos sobre as arquiteturas orientadas a serviços (SOA) que são uma forma de desenvolvimento de sistemas distribuídos em que os componentes de sistema são serviços autônomos, executando em computadores geograficamente distribuídos. E que os serviços são plataforma e implementação independentes de linguagem e da aplicação que o usa e que a engenharia de serviços é o processo de desenvolvimento de serviços para reuso em aplicações orientadas a serviços.

E por fim, falaremos sobre as tendências leves e como as novas tecnologias são introduzidas regularmente, apresentadas como sendo a “solução” para muitos dos problemas que os engenheiros de software enfrentam e incorporadas nos projetos grandes e pequenos. Estamos preparados para as novas tecnologias? Você está preparado, como futuro engenheiro de software para as novas tecnologias?

Lembre-se: coloque em prática o que aprendeu ao longo do livro, estude, pratique e pesquise e procure sempre testar diferentes abordagens, até encontrar a que funciona melhor para você, sua equipe e seus clientes.

Desejamos sucesso em todos os seus projetos!

ATIVIDADES



1. Segundo Sommerville (2011), praticamente todos os grandes sistemas computacionais agora são sistemas distribuídos, ou seja, um sistema que envolve vários computadores, em contraste com sistemas centralizados, em que todos os componentes do sistema executam em um único computador. Com base nesta informação, analise as assertivas a seguir:

- I. Um sistema distribuído é um conjunto de computadores dependentes que se apresenta a seus usuários como um sistema único e coerente.
- II. Um sistema distribuído consiste em componentes autônomos.
- III. Um sistema distribuído é aquele no qual os componentes localizados em computadores interligados em rede se comunicam e coordenam suas ações apenas passando mensagens.
- IV. Os sistemas distribuídos são, inherentemente, mais complexos que os sistemas centralizados, o que os torna mais difíceis para projetar, implementar e testar.

Assinale a alternativa correta:

- a) I, II apenas.
- b) II, III apenas.
- c) I, II, III apenas.
- d) II, III, IV apenas.
- e) I, II, III, IV apenas.

2. Conforme Sommerville (2011, p. 341), os projetistas de sistemas distribuídos precisam organizar seus projetos de sistema para encontrar um equilíbrio entre desempenho, confiança, proteção e capacidade de gerenciamento do sistema. Pensando nisso, qual a diferença entre o modelo cliente-magro e cliente-gordo?

ATIVIDADES



3. Segundo Sommerville (2011, p. 338), “um sistema distribuído requer um software que possa gerenciar essas diversas partes e assegurar que elas podem se comunicar e trocar dados”. O termo ‘middleware’ é usado para se referir a esse software — ele fica no meio, entre os componentes distribuídos do sistema. Com base nisso, assinale a alternativa que possui exemplos de middleware:
- I. Software para Gerenciamento de Comunicações com Bancos de Dados.
 - II. Gerenciadores de Transações.
 - III. Conversores de Tipos de fontes.
 - IV. Controladores de Comunicação.
- Assinale a alternativa correta:
- a) I, II apenas.
 - b) II, III apenas.
 - c) I, II, III apenas.
 - d) I, II, IV apenas.
 - e) I, II, III, IV apenas.
4. Segundo Sommerville (2011), um serviço pode ser definido como um componente de software de baixo acoplamento, reusável, que encapsula funcionalidade discreta, que pode ser distribuída e acessada por meio de programas. Com base nesta informação, descreva os três aspectos de um Web service com a especificação WSDL?

ATIVIDADES



5. O sistema é complexo e que não pode ser controlado pelos proprietários dos sistemas que usam a rede. Para Sommerville (2011), portanto, essa é uma imprevisibilidade inerente à operação de sistemas distribuídos, a qual deve ser considerada pelo projetista do sistema. Sobre a imprevisibilidade, analise as alternativas e assinale a correta:

- a) Transparência, Abertura , Portabilidade, Proteção, Qualidade de serviço, Gerenciamento de Falhas.
- b) Transparência, Abertura, Escalabilidade, Proteção, Qualidade de Testes, Gerenciamento de Riscos.
- c) Transparência, Fechamento, Escalabilidade, Proteção, Qualidade de serviço, Gerenciamento de Riscos.
- d) Transparência, Abertura, Escalabilidade, Proteção, Qualidade de serviço, Gerenciamento de Falhas.
- e) Transparência, Fechamento, Escalabilidade, Confiança, Qualidade de serviço, Gerenciamento de Falhas.



DESENVOLVIMENTO DE SOFTWARE ORIENTADO A ASPECTOS

Independentemente do processo de software escolhido, os desenvolvedores de software complexos, invariavelmente, implementam um conjunto de recursos, funções e conteúdo localizados. Essas características de software localizadas são modeladas como componentes (por exemplo, classes orientadas a objetos) e, em seguida, construídas dentro do contexto da arquitetura do sistema. À medida que os modernos sistemas baseados em computadores se tornam mais sofisticados (e complexos), certas *restrições* — propriedades exigidas pelo cliente ou áreas de interesse técnico — se estendem por toda a arquitetura. Algumas restrições são propriedades de alto nível de um sistema (por exemplo, segurança, tolerância a falhas). Outras afetam funções (por exemplo, a aplicação de regras de negócio), sendo que outras são sistêmicas (por exemplo, sincronização de tarefas ou gerenciamento de memória).

Quando restrições cruzam múltiplas funções, recursos e informações do sistema, elas são, frequentemente, denominadas *restrições cruzadas*. Os *requisitos de aspectos* definem as restrições cruzadas que têm um impacto por toda a arquitetura de software. O desenvolvimento de software orientado a aspectos AOSD, *Aspect-Oriented Software Development*), com frequência conhecido como programação orientada a aspectos (AOP, *Aspect-Oriented Programming*), é um paradigma de engenharia de software relativamente novo que oferece uma abordagem metodológica e de processos para definir, especificar, projetar e construir *aspectos* — “mecanismos além das sub-rotinas e herança para localizar a expressão de uma restrição cruzada”. A AOCE usa um conceito de fatias horizontais por meio de componentes de software decompostos verticalmente, chamados “*aspectos*”, para caracterizar propriedades funcionais ou não funcionais cruzadas dos componentes. Aspectos sistêmicos, comuns, incluem interfaces com o usuário, trabalho colaborativo, distribuição, persistência, gerenciamento de memória, processamento de transações, segurança, integridade e assim por diante. Os componentes podem fornecer ou requerer um ou mais “detalhes de aspecto” relativo a um determinado aspecto, tais como um mecanismo de visualização, exequibilidade extensível e gênero de interface (aspectos da interface com o usuário); geração de eventos, transporte e recebimento (aspectos de distribuição); armazenamento/recuperação e indexação de dados (aspectos de persistência); autenticação, codificação e direitos de acesso (aspectos de segurança); atomicidade de transações, controle de concorrência e estratégia de entrada no sistema (aspectos transacionais) e assim por diante. Cada detalhe de aspecto possui uma série de propriedades relativas a características funcionais e não funcionais do detalhe do aspecto.

Um processo distinto orientado a aspectos ainda não atingiu sua maturação. Entretanto, é provável que um processo desses irá adotar características tanto dos modelos de processo evolucionário quanto de concorrente. O modelo evolucionário é apropriado quando os aspectos são identificados e então construídos. A natureza paralela do desenvolvimento concorrente é essencial, porque os aspectos são criados independentemente de componentes de software localizado e, apesar disso, os aspectos têm um impacto direto sobre esses componentes. Portanto, é essencial instanciar comunicação assíncrona entre as atividades de processos de software aplicadas na engenharia e construção de aspectos e componentes.

Fonte: Pressman (2016, p. 69 -71).





SEPARAÇÃO DE INTERESSES

A separação de interesses é um princípio-chave de projeto e implementação de software. Isso significa que você deve organizar seu software de tal forma que cada elemento do programa (classe, método, procedimento etc.) faça apenas uma coisa. Assim poderá concentrar-se nesse elemento sem se preocupar com outros elementos no programa. Poderá entender cada parte do programa conhecendo seus interesses, sem a necessidade de entender outros elementos. Quando as mudanças forem necessárias, elas serão feitas em um número pequeno de elementos.

A importância de separação de interesses foi reconhecida nos primeiros estágios da história da ciência da computação. Subrotinas, que encapsulam uma unidade de funcionalidade, foram inventadas no início da década de 1950 e mecanismos de estruturação de programas posteriores, como procedimentos e classes de objeto, foram projetados para prover mecanismos melhores para realizar a separação de interesses. No entanto, todos esses mecanismos possuem problemas em lidar com certos tipos de interesses que cruzam outros interesses. Esses interesses transversais não podem ser localizados por meio de mecanismos estruturados como objetos e funções. Aspectos foram inventados para ajudar a gerenciar esses interesses transversais. Embora seja aceito de forma generalizada que a separação de interesses é uma boa prática de engenharia de software, é mais difícil definir o que de fato significa um interesse nesse contexto. Às vezes, é definido como uma noção funcional (por exemplo, um interesse é algum elemento ou funcionalidade em um sistema). Como alternativa, ele pode ser definido muito amplamente como ‘qualquer parte de interesse ou foco de um programa’. Nenhuma dessas definições é muito útil na prática; os interesses são certamente mais que simples elementos funcionais, porém a definição mais generalizada é tão vaga que é praticamente inútil.

O desempenho de sistema pode ser um interesse, porque os usuários querem uma resposta rápida do sistema; alguns stakeholders podem estar interessados em que o sistema inclua determinada funcionalidade; pode ser interessante a empresas que dão suporte ao sistema que o sistema seja fácil de manter. Dessa forma, um interesse pode ser definido como algo que é importante ou significativo para um stakeholder ou um grupo de stakeholders.

Existem vários tipos diferentes de interesses de stakeholder:

1. Interesses funcionais, relacionados com uma funcionalidade específica a ser incluída no sistema. Por exemplo, em um sistema de controle de trens, um interesse funcional específico seria a frenagem do trem.
2. Interesses de qualidade de serviço, relacionados ao comportamento não funcional de um sistema. Incluem características como desempenho, confiabilidade e disponibilidade.
3. Interesses de políticas, relacionados com as políticas gerais que governam o uso de um sistema. Interesses de políticas incluem interesses de segurança e interesses relacionados com regras de negócio.



4. Interesses de sistemas, relacionados com atributos do sistema como um todo, tais como manutenibilidade e configurabilidade.
5. Interesses organizacionais, relacionados com objetivos e prioridades organizacionais. Incluem a produção de um sistema dentro do orçamento, fazendo uso de ativos existentes de software e mantendo a reputação da organização.

Os interesses centrais de um sistema são aqueles interesses funcionais que se relacionam com seu propósito principal. Portanto, para um sistema hospitalar de informações de pacientes, por exemplo, os interesses funcionais centrais são a criação, a edição, a consulta e o gerenciamento de registros de pacientes. Além dos interesses centrais, os grandes sistemas possuem também interesses funcionais secundários. Estes podem envolver funcionalidade que compartilha informações com os interesses centrais, ou que é requerido para que o sistema possa satisfazer seus requisitos não funcionais.

Fonte: Sommerville (2011, p. 396 - 398).

MATERIAL COMPLEMENTAR



LIVRO

SOA - Modelagem, Análise e Design

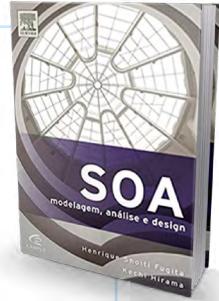
Kechi Hirama, Henrique Fugita

Editora: Elsevier

Sinopse: com o intuito de automatizar atividades e aumentar a produtividade, as empresas vêm empregando cada vez mais sistemas de TI no suporte ao negócio, tais como ERP, para controlar sua operação; CRM, para gerenciar informações sobre o relacionamento com seus clientes; e SCM, para gerenciar interações com parceiros, fornecedores e consumidores.

Para atender essa infraestrutura, em que cada sistema possui um conjunto de dados e informações próprio, vem se desenvolvendo e sendo aceito o conceito de Arquitetura Orientada a Serviços (SOA), um paradigma para organizar e utilizar competências distribuídas entre vários domínios.

Neste livro, os autores apresentam a SOA como um modo de estruturar as aplicações de software em elementos chamados serviços, seguindo um conjunto de princípios alinhados aos requisitos de negócio.



LIVRO

SOA na Prática

Fabio Perez Marzullo

Editora: Novatec

Sinopse: este livro apresenta SOA (Service-Oriented Architecture – Arquitetura Orientada a Serviços) como uma disciplina emergente capaz de relacionar serviços de negócios com serviços de TI. Descreve os principais conceitos que norteiam a teoria de serviços, como definição, alinhamento estratégico, e-business, arquiteturas, entre outros, e as principais técnicas utilizadas na construção de soluções SOA, como Web Services, padrões, qualidade de serviço e segurança. Mostra como soluções SOA são utilizadas na prática, descrevendo técnicas de aplicação da teoria e estudos de caso de projetos reais. Oferece ferramentas para que o leitor possa dar os primeiros passos rumo à criação de soluções verdadeiramente alinhadas a serviços e o incita a pensar melhor sobre como planejar suas soluções baseando-se nos recursos que lhe são disponibilizados por sua organização. Pode ser usado como obra de apoio a estudantes, pesquisadores e profissionais na exploração e pesquisa de novas técnicas que utilizem SOA como solução para alinhamento entre o negócio e a TI.



MATERIAL COMPLEMENTAR



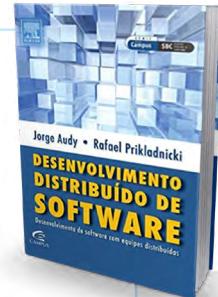
LIVRO

Desenvolvimento Distribuído de Software

Jorge Audy, Rafael Prikladnicki

Editora: Elsevier

Sinopse: este livro busca analisar e apresentar os principais conceitos na área da Engenharia de Software, conhecida como DDS (Desenvolvimento Distribuído de Software) ou GSD (Desenvolvimento Global de Software). DDS apresenta características próprias, multifacetadas, que devem ser bem entendidas pelos pesquisadores e profissionais que atuam no mercado de desenvolvimento de software, tanto na área teórica como na prática das organizações.



LIVRO

Engenharia de Software. Qualidade e Produtividade com Tecnologia

Kechi Hirama

Editora: Elsevier

Sinopse: este livro apresenta uma visão moderna e inovadora do conceito de Engenharia de Software. Em vez de explorar à exaustão tópicos tradicionais da disciplina, como fazem os títulos considerados clássicos da área, a obra dá ênfase aos conceitos requisitados pelos entrantes nesse universo, tais como documentação de processos, novas formas de padrões arquiteturais, gerência de projetos OO, gerência de requisitos e, por fim, o novíssimo manifesto ágil (Scrum) e os projetos e implementação de serviços SOA.

A importância do livro se concretiza pelo fato de apresentar todos os temas de maneira direta e totalmente voltada para a realidade dos estudantes e professores brasileiros.



MATERIAL COMPLEMENTAR



NA WEB

Afinal o que é um sistema distribuído?

Recomendo um artigo a respeito dos sistemas distribuídos estarem em todo o lugar, ou melhor, acessíveis a partir de qualquer lugar. Um sistema distribuído é um conjunto de computadores independentes entre si (e até diferentes), ligados por meio de uma rede de dados, que se apresentam aos utilizadores como um sistema único e coerente. Para saber mais, acesse o link: <<https://pplware.sapo.pt/informacao/afinal-o-que-e-um-sistema-distribuido/>>.



NA WEB

SOA – Arquitetura Orientada a Serviços

Sugiro também outro artigo sobre SOA, seus conceitos antigos e modernos. Ele inicia falando sobre a pergunta: o que é SOA? Acesse o link para conhecer: <<http://blog.iprocess.com.br/2012/10/soa-arquitetura-orientada-a-servicos/>>.



NA WEB

Vantagens e Desvantagens de SOA

Veja neste artigo uma apresentação da Arquitetura Orientada a Serviços (SOA), as vantagens e desvantagens em utilizá-la em projetos. Para ficar por dentro, acesse o link: <<https://www.devmedia.com.br/vantagens-e-desvantagens-de-soa/27437>>.

REFERÊNCIAS

COULOURIS, G. et al.; **Sistemas Distribuídos** - Conceitos e Projeto. 5. ed. Porto Alegre: Bookman, 2013.

PRESSMAN, R.; MAXIM, B. R. **Engenharia de Software** – Uma abordagem profissional. 8. Ed. Porto Alegre: AMGH, 2016.

SOMMERVILLE, I. **Engenharia de Software**. 9.ed. - São Paulo: Pearson Prentice Hall, 2011.

TANENBAUM, A. S.; VAN STEEN, M. **Sistemas Distribuídos** - Princípios e Paradigmas. 2. ed. São Paulo: Pearson Prentice Hall, 2007.



GABARITO

1. Opção correta é a D.
2. No Modelo cliente-magro a camada de apresentação é implementada no cliente e todas as outras camadas (gerenciamento de dados, processamento de aplicação e banco de dados) são implementadas em um servidor. Já no Modelo cliente-gordo: é a parte ou todo o processamento de aplicação é executado no cliente e as funções de banco de dados e gerenciamento são implementadas no servidor.
3. Opção correta é a D.
4. Os três aspectos de um web service são: o que faz o serviço, como ele se comunica e onde o encontrar. **O que** de um documento WSDL, denominado interface, especifica quais operações o serviço suporta e define o formato das mensagens que são enviadas e recebidas pelo serviço. **O como** de um documento WSDL, denominado ligação, mapeia a interface abstrata para um conjunto concreto de protocolos. A ligação especifica os detalhes técnicos de como se comunicar com um web service. **Onde** de um documento WSDL descreve o local da implementação de um web service específico (seu ponto final).
5. Opção correta é a D.



CONCLUSÃO

Caro(a) aluno(a), assim terminamos nossa jornada. Foram cinco unidades da disciplina de Tópicos Especiais. Espero que sua leitura tenha sido agradável e que o conteúdo visto tenha contribuído para seu crescimento pessoal e profissional.

Nesta disciplina, abordamos vários conteúdos sobre o problema de entregar software, o desenvolvimento de Sistemas Críticos, a Reengenharia e a Engenharia Reversa, reutilização de software, a refatoração para padrões e as tendências emergentes da engenharia de software.

Na unidade I, foi feita uma apresentação geral sobre os princípios da Entrega Contínua de Software, os problemas da entrega e quais as práticas necessárias para realizá-la. Após, seguimos para a unidade II, onde foi apresentada uma visão geral sobre os Sistemas Críticos, Engenharia de Segurança, Reengenharia e Manutenção e Engenharia Reversa. Nesta unidade aprendemos conceitos importantes sobre o porque a confiança é considerada a propriedade mais importante em um sistema crítico. Na unidade III, foi apresentado os principais conceitos sobre o Reuso de Software e as suas vantagens e desvantagens. Falamos sobre o Reuso de Componentes e os Frameworks e a Engenharia de Software baseada em Componentes.

Na sequência, seguimos para a unidade IV, onde foi introduzido os conceitos e princípios sobre a refatoração para padrões. Aprendemos como a refatoração pode ser aplicada de maneira controlada e eficiente, que não introduza erros no código e que melhore metódicamente a sua estrutura do sistema que está em desenvolvimento.

E por fim, na unidade V, aprendemos sobre as Tendências Emergentes da Engenharia de Software e como essas tendências que têm um efeito sobre a tecnologia de engenharia de software, seus cenários de negócios, organizacionais, mercado e cultural. Também falamos sobre Sistemas Distribuídos e arquiteturas orientadas a serviços (SOA).

Espero que você coloque em prática tudo o que aprendeu ao longo das unidades e continue estudando, praticando e pesquisando. Assim, não pare por aqui! Desejamos a você muito sucesso, sempre!



ANOTAÇÕES



ANOTAÇÕES

