UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

CAMPUS CAMPO MOURÃO

RESENHA CAPÍTULO 2 E 9 - LIVRO "Martin. Fowler. UML Essencial: Um Breve Guia para Linguagem Padrão. Martin Fowler; trad. João Tortello.– 3.ed. Porto Alegre: Bookman, 2005. ISBN: 8536304545."

1. Processo de Desenvolvimento

Logo de inicio, o livro nos garante a ideia que, a maneira como utilizaremos a **UML** dependerá de qual processo utilizarmos.

O autor afirma que prefere utilizar o iterativo, porém, necessitando ser bem planejado para expor cedo os riscos e obter um melhor controle sobre o desenvolvimento. Sendo assim, apresentamos os processos mostrados no livro.

1.1 Processo Iterativo e Em Cascada

Quando estamos abordando estes dois tipos de processos, a principal diferença entre eles, é a maneira como subdividimos o projeto em segmentos menores.

1.1.1 Em Castata

Onde dividimos o projeto em atividades, por exemplo, nosso projeto poderia ter uma fase de dois meses de análise, seguida de um fase de projeto de quatro meses, três meses de codificação e três meses de testes, totalizando um ano inteiro construção de um software.

Destacamos que, podem ocorrer necessidades de revisões das decisões de análise e projeto em etapas posteriores, porém, deve ocorrer o mínimo de vezes possível.

1.1.2 Iterativo

Neste dividimos o projeto em subconjuntos de funcionalidades, ou seja, dado 12 meses, subdividimos em 4 iterações. Sendo que em cada uma, concluímos o ciclo de vida do software referente a sua quantidade de requisitos ($\frac{1}{4}$ de requisitos para cada itereção).

Ao final de cada iteração, o ideal seria termos um software com qualidade de produção, porém, na maioria dos casos, necessitamos de um período para remover erros.

Em momentos onde percebemos que não iremos conseguir finalizar todos os requisitos na iteração atual, utilizamos a técnica de **quadro de tempo**. Deslocando algumas funcionalidades para iterações posteriores.

1.2 Planejamento Preditivo e Adaptativo

Utilizamos o planejamento preditivo quando, realizamos o planejamento completo do software antes do processo começar, com objetivo de gerar maior entendimento posteriormente. Sendo dividido em duas etapas: a primeira sugere planos e é difícil prever, enquanto a segunda se torna mais previsível, pois os planos estão definidos.

Um grande problema enfrentado pelo planejamento preditivo, é a revolução dos requisitos. Onde necessitamos realizar alterações, em etapas avançadas do projeto. Uma solução seria não permitir que as alterações sejam concluídas, porém, o software final pode não atender a demanda do cliente.

Ao trabalharmos com o planejamento adaptativo, é obrigatório o uso do processo iterativo, pois precisaremos realizar algumas alterações durante a construção do *software*.

Desta forma, apresentamos duas recomendações extremamente válidas:

1. Não faça um plano preditivo até ter todos os requisitos precisos e exatos e até estar confiante de que eles não mudarão significativamente.

2. Se não obter requisitos precisos e estáveis, utilize o planejamento adaptativo.

1.3 Processos Ágeis

Os processos ágeis, primam que o sucesso de um projeto é devido a qualidade das pessoas que estão envolvidas nele e o quão bem trabalham juntas.

As metodologias utilizam iterações curtas, frequentemente de um mês ou menos, desprezando o uso da **UML** no projeto. Os mesmos tendem a ter pouca formalidade, afirmando que o uso dela torna as alterações mais difíceis, indo contra a natureza das pessoas talentosas.

1.4 *Rational Unified Process*

RUP trata-se de uma estrutura de processos, que fornece um vocabulário e uma vaga estrutura para falar sobre os processos. Em todos os casos de desenvolvimento, o **RUP** é um processo interativo. Todos os projetos **RUP** utiliza as quatro fases a seguir (temos algumas imprecisões sobre as fases):

1. A **concepção** faz uma analise inicial do projeto.
2. A **elaboração** identifica os casos de usos principais do projeto e elabora o software em iterações, tendo no final uma boa ideia dos requisitos e um esqueleto funcional do sistema.
3. A **construção** continua o processo, desenvolvendo funcionalidades suficientes para o lançamento.
4. A **transição** inclui atividades que não realizamos no processo de forma iterativa.

1.5 Como adequar um processo a um projeto?

Não teremos um processo único que funcione para o desenvolvimento de todos os *softwares*, sendo assim, em alguns casos necessitaremos adaptar processos.

O livro aborda que, em desenvolvedores em experiências, é considerável utilizar um processo pronto, aprendendo como o mesmo funciona, para somente depois conseguir realizar alterações.

Um bom método de adaptação dos processos, é realizar **retrospectivas de iteração**. Onde reunimos toda a equipe a fim de considerar falhas e como devem ser aprimoradas. Em reuniões mais curtas, podemos utilizar uma lista com três categorias.

- **Manter:** as coisas que funcionaram bem.
- **Problemas:** áreas que não estão funcionando bem.
- **Tentativa:** alterações para aprimorar o processo.

Durante nosso processo, utilizamos a UML para transmitir as ideias com mais clareza.

1.6 Análise de Requisito

Nesta etapa descobrimos o que o usuário e o cliente do software quer que ele faça, podemos violar algumas regras da **UML** para facilitar a comunicação com usuários e clientes, caso necessário. Veja abaixo algumas técnicas **UML** utilizadas durante este processo.

- Casos de uso, que descreve como as pessoas interagem com o sistema.
- Um diagrama de classe, sendo uma boa maneira para construir um vocabulário rigoroso do domínio.
- Um diagrama de atividades, mostrando o fluxo de como o software e as atividades humanas interagem.
- Um diagrama de estados, usado para mostrar as mudanças de estados que um objeto pode sofrer.

1.7 Projeto

Quando estamos projetando um sistema, existe alguns diagramas **UML** mais técnicos que nos auxiliam bastante, sendo eles:

- **Diagrama de Classe** mostra as classes presentes no código e como elas se relacionam.
- **Diagrama de Sequência** para cenários mais comuns.
- **Diagrama de Pacote** mostra a organização em larga escala do software.
- **Diagrama de Estado** para classes com históricos de vidas complexos.
- **Diagrama de Distribuição** para mostrar o layout físico do software.

Muitas dessas técnicas podem ser utilizadas para documentar um software, quando já codificado. Esperamos também que, a implementação do código acompanhe o projeto **UML** feito, porém, em alguns casos necessitaremos realizar alterações em nossa modelagem.

1.8 Documentação

O autor do livro cita que, podemos utilizar diagramas **UML** para complementar a documentação de um software. Porém, precisamos construir uma documentação detalhada a partir do código de nosso sistema. Veja alguns exemplos de utilização de diagramas em documentações.

- **Diagrama de Pacotes:** ajuda a compreender as partes lógicas do sistemas, ver as dependências e mantê-las sobre controle.
- **Diagrama de Classe:** dentro do diagrama de pacote, o autor utiliza os diagramas de classe como um sumário gráfico.
- **Diagrama de Máquina de Estado:** caso nossas classes tenham um comportamento de ciclo de vida complexo, utilizamos este tipo de diagrama para descrevê-lo.

Devemos também documentar as medidas de projetos que não adotamos, e o motivo de não termos aplicado as mesmas.

1.9 Como entender um código legado?

Podemos utilizar diagramas para conseguir entender um emaranhado de código, uma boa estrategia é utilizar diagramas de sequência para ver como vários objetos colaboram no tratamento de um método complexo.

2. Casos de Uso

Os casos de uso fornecem uma narrativa de como nosso software é utilizado, ou seja, descreve as interações típicas entre os usuários do sistema.

Segundo o autor, é mais fácil descrever os casos de uso utilizando cenários. Um cenário é uma sequência de passos que descreve uma interação entre um usuário e um sistema (descrevendo todos os casos possíveis).

2.1 Conteúdo de um caso de uso

Cada caso de uso possui um ator principal (usuário) que pede para o sistema executar um serviço. Podem existir outros atores que nosso sistema se comunica enquanto executa o caso de uso, eles são chamados de atores secundários.

Em cada passo de nosso caso de uso, devemos declarar de forma simples, mostrando claramente quem está executando determinado passo (cada passo demonstra a intenção do ator).

Uma extensão dentro do caso de uso mostra diferentes cenários que nosso software pode tomar, para isso, devemos nos perguntar em cada passo: “Como isso poderia ser

feito de uma outra forma? O que poderia dar errado?". Veja na Figura 2.1 um exemplo onde temos um texto de caso de uso e suas respectivas extensões.

Compra de um Produto

Nível do Objetivo: Nível do Mar

Cenário Principal de Sucesso:

1. O cliente navega pelo catálogo e seleciona itens para comprar
2. O cliente vai para o caixa
3. O cliente preenche o formulário da remessa (endereço de entrega; opção de entrega imediata ou em três dias)
4. O sistema apresenta a informação completa do faturamento, incluindo a remessa
5. O cliente preenche a informação de cartão de crédito
6. O sistema autoriza a compra
7. O sistema confirma imediatamente a venda
8. O sistema envia uma confirmação para o cliente por *e-mail*

Extensões:

3a: Cliente regular
.1: O sistema mostra a informação atual da remessa, a informação de preço e a informação de cobrança
.2: O cliente pode aceitar ou escrever por cima desses padrões, retornando ao CPS, no passo 6

6a. O sistema falha na autorização da compra a crédito
.1: O cliente pode inserir novamente a informação do cartão de crédito ou cancelar

Figura 2.1: Exemplo texto de caso de uso.

Podemos ter um caso de uso incluído em outro, ou seja, um passo complicado de um caso de uso, pode ser um outro caso de uso. Esta subdivisão é útil em situações onde, um passo complexo congestionaria nosso sistema principal, ou em passos repetidos em vários casos de uso. Desta forma, podemos adicionar algumas informações importantes em cada caso de uso.

- **Pré-Condição:** informa o que o sistema deve garantir como verdadeiro antes de começar o caso de uso.
- **Garantia:** descreve o que o sistema deve assegurar no fim de cada caso. As garantias de sucesso se mantem em cenários bem-sucedidos, e as mínimas após qualquer cenário.
- **Gatilho:** especifica o evento que inicia o caso de uso.

Lembre-se de sempre tentar manter cada caso de uso breve e fácil de ler.

2.2 Diagramas de Casos de Uso

A **UML** não nos diz nada sobre o conteúdo de um caso de uso, porém, nos fornece um formato de diagrama para mostrá-lo. Veja na Figura 2.2 um exemplo de diagrama de caso de uso.

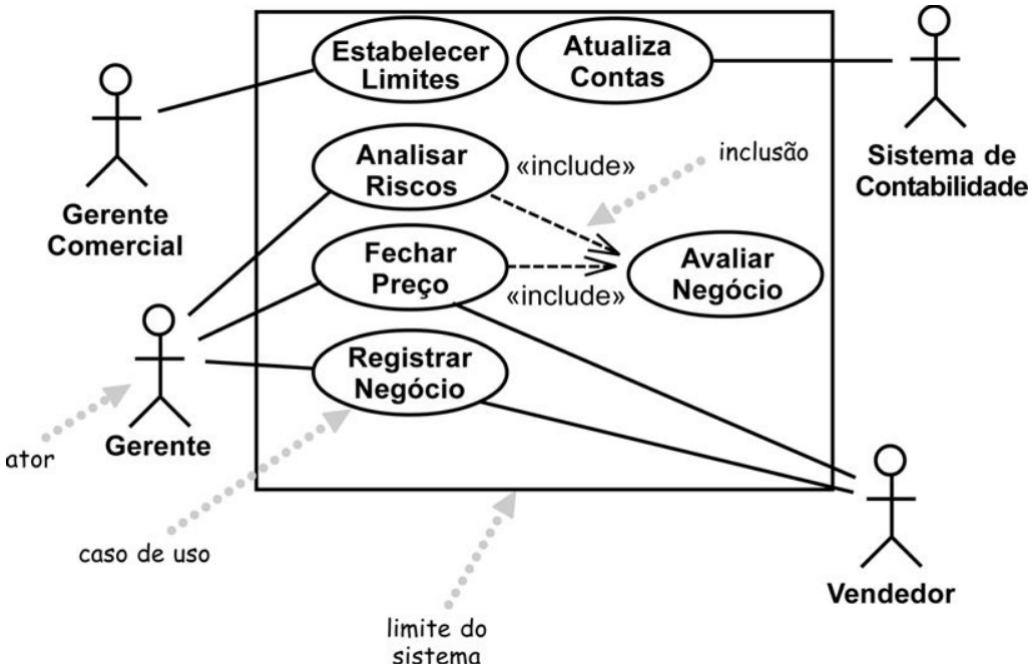

Figura 2.2: Diagrama caso de uso.

O diagrama acima serve como ferramenta de sumário gráfico, ou seja, conseguimos mostrar os atores, os casos de uso e os relacionamento entre eles.

2.3 Caso de Uso e Funcionalidades

As funcionalidades constituem uma boa maneira de repartir um sistema para planejarmos de maneira iterativa. Enquanto os casos de uso servem para descrever em forma narrativa como os atores (usuários) utilizam o sistema.

2.4 Quando utilizar casos de uso?

Primeiramente, devemos deixar claro que, o caso de uso representa um visão externa do sistema, não tendo nenhuma correlação com as classes do mesmo.

Desta forma, devemos montar nossos casos de usos com fácil legibilidade, intencionado a ajudar o entendimento dos requisitos funcionais do sistema. Algumas versões detalhados poderão ser feitas antes do desenvolvimento desse caso de uso.

Concentraremos nossas energias no modo texto do caso de uso, pois o mesmo contém todo valor da técnica necessária.