Projeto Orientado a Objetos

Prof[a]. Simone Cristina Aléssio

UNIASSELVI
2016

Elaboração:
Profª. Simone Cristina Aléssio

Revisão, Diagramação e Produção:
Centro Universitário Leonardo da Vinci – UNIASSELVI

Ficha catalográfica elaborada na fonte pela Biblioteca Dante Alighieri
UNIASSELVI – Indaial.

A372p

Aléssio; Simone Cristina

Projeto orientado a objetos/ Simone Cristina Aléssio : UNIASSELVI, 2016.

196 p.; il.

ISBN 978-85-7830-976-3

1.Programação Orientada a Objetos. I. Centro Universitário Leonardo Da Vinci.

CDD 005.1

Impresso por:

Apresentação

Seja bem-vindo ao mundo de Projeto Orientado a Objetos. Neste universo você vai se deparar com termos como atributos, métodos, abstração, encapsulamento, classes, hierarquia das classes, processo unificado, entre outros, que compõem o material de estudo desta disciplina e, por consequência, o dia a dia de um analista, desenvolvedor, programador, ou seja, o profissional da programação.

Este caderno pressupõe que você já possua alguma experiência anterior em programação, principalmente JAVA, e que já conheça os conceitos de Gestão de Projetos. Ou seja, mostra-se imprescindível o uso dos conhecimentos adquiridos em disciplinas anteriores para a compreensão da disciplina de Projeto Orientado a Objetos.

Destacamos a importância de desenvolver as autoatividades, afinal, cada exercício deste caderno foi desenvolvido para a fixação de conceitos por meio da prática. Em caso de dúvidas, entre em contato com o seu tutor externo ou com a equipe da tutoria da Uniasselvi, evitando assim, o prosseguimento das atividades sem sanar todas as dúvidas que porventura surgirem.

Vale destacar a necessidade de dedicação e de muita determinação, afinal a Programação Orientada a Objetos exige de você bem mais do que apenas este caderno para sua total compreensão. Aqui, você recebe somente um início, ou seja, os conceitos de determinados pontos importantes na programação, porém é somente na prática que você consegue compreender o mundo da programação como um todo.

Lembre-se de que o mundo da informática é encantador, assim, seu entusiasmo por este universo depende somente de você, ressaltamos neste momento a compreensão da lógica envolvida no processo de construção de programas. Por este motivo, destacamos uma frase que consideramos importante no caso da programação, afinal: "Para se ter sucesso, é amar de verdade o que se faz. Caso contrário, levando em conta apenas o lado racional, você simplesmente desiste. É o que acontece com a maioria das pessoas" (Steve Jobs – criador da Apple).

Bom estudo! Sucesso na sua trajetória acadêmica e profissional!

Profª. Simone Cristina Aléssio

Você já me conhece das outras disciplinas? Não? É calouro? Enfim, tanto para você que está chegando agora à UNIASSELVI quanto para você que já é veterano, há novidades em nosso material.

Na Educação a Distância, o livro impresso, entregue a todos os acadêmicos desde 2005, é o material base da disciplina. A partir de 2017, nossos livros estão de visual novo, com um formato mais prático, que cabe na bolsa e facilita a leitura.

O conteúdo continua na íntegra, mas a estrutura interna foi aperfeiçoada com nova diagramação no texto, aproveitando ao máximo o espaço da página, o que também contribui para diminuir a extração de árvores para produção de folhas de papel, por exemplo.

Assim, a UNIASSELVI, preocupando-se com o impacto de nossas ações sobre o ambiente, apresenta também este livro no formato digital. Assim, você, acadêmico, tem a possibilidade de estudá-lo com versatilidade nas telas do celular, *tablet* ou computador.

Eu mesmo, UNI, ganhei um novo *layout*, você me verá frequentemente e surgirei para apresentar dicas de vídeos e outras fontes de conhecimento que complementam o assunto em questão.

Todos esses ajustes foram pensados a partir de relatos que recebemos nas pesquisas institucionais sobre os materiais impressos, para que você, nossa maior prioridade, possa continuar seus estudos com um material de qualidade.

Aproveito o momento para convidá-lo para um bate-papo sobre o Exame Nacional de Desempenho de Estudantes – ENADE.

Bons estudos!

Olá acadêmico! Para melhorar a qualidade dos materiais ofertados a você e dinamizar ainda mais os seus estudos, a Uniasselvi disponibiliza materiais que possuem o código *QR Code*, que é um código que permite que você acesse um conteúdo interativo relacionado ao tema que você está estudando. Para utilizar essa ferramenta, acesse as lojas de aplicativos e baixe um leitor de *QR Code*. Depois, é só aproveitar mais essa facilidade para aprimorar seus estudos!

BATE PAPO SOBRE O ENADE!

Olá, acadêmico!

Você já ouviu falar sobre o **ENADE**?

Se ainda não ouviu falar nada sobre o ENADE, agora você receberá algumas informações sobre o tema.

Ouviu falar? Ótimo, este informativo reforçará o que você já sabe e poderá lhe trazer novidades.

Vamos lá!

Qual é o significado da expressão ENADE?

EXAME NACIONAL DE DESEMPENHO DOS ESTUDANTES

Em algum momento de sua vida acadêmica você precisará fazer a prova ENADE.

Que prova é essa?

É **obrigatória**, organizada pelo INEP – Instituto Nacional de Estudos e Pesquisas Educacionais Anísio Teixeira.

Quem determina que esta prova é obrigatória... O **MEC – Ministério da Educação**.

O objetivo do MEC com esta prova é o de avaliar seu desempenho acadêmico assim como a qualidade do seu curso.

Fique atento! Quem não participa da prova fica impedido de se formar e não pode retirar o diploma de conclusão do curso até regularizar sua situação junto ao MEC.

Não se preocupe porque a partir de hoje nós estaremos auxiliando você nesta caminhada.

Você receberá outros informativos como este, complementando as orientações e esclarecendo suas dúvidas.

Você tem uma trilha de aprendizagem do ENADE, receberá e-mails, SMS, seu tutor e os profissionais do polo também estarão orientados.

Participará de webconferências entre outras tantas atividades para que esteja preparado para #mandar bem na prova ENADE.

Nós aqui no NEAD e também a equipe no polo estamos com você para vencermos este desafio.

Conte sempre com a gente, para juntos mandarmos bem no ENADE!

UNIASSELVI

SUMÁRIO

UNIDADE 1

CONCEITO DE PROJETOS EM OO; FASES E CARACTERÍSTICAS; UMA VISÃO GERAL DE PROJETOS ORIENTADOS A OBJETOS; PROJETO DE SOFTWARE ORIENTADO A OBJETOS

OBJETIVOS DE APRENDIZAGEM

A partir desta unidade, você será capaz de:

- conhecer os principais conceitos de projetos orientados a objetos, bem como as suas principais características;
- rever conceitualmente processos de desenvolvimento de software orientado a objetos;
- aprimorar os conhecimentos acerca da análise, projeto e desenvolvimento de software orientado a objetos;
- conhecer os conceitos de desenvolvimento Orientado a Objetos e o Processo Unificado.

PLANO DE ESTUDOS

Esta unidade de ensino contém três tópicos, sendo que no final de cada um, você encontrará atividades que contribuirão para a apropriação dos conteúdos.

TÓPICO 1 – CONCEITO E CARACTERÍSTICAS DE PROJETOS ORIENTADOS A OBJETOS

TÓPICO 2 – VISÃO GERAL DA CONSTRUÇÃO DE UM PROJETO ORIENTADO A OBJETOS; FASES E RESPECTIVAS ATIVIDADES

TÓPICO 3 – PROJETO DE SOFTWARE ORIENTADO A OBJETOS; DECISÕES EM PROJETOS ORIENTADOS A OBJETOS

TÓPICO 1

CONCEITO E CARACTERÍSTICAS DE PROJETOS ORIENTADOS A OBJETOS

1 INTRODUÇÃO

Projeto é algo temporário, com sequência de atividades com início, meio e fim. Seu resultado final fornece um produto ou serviço único e progressivo, tangível ou intangível restrito a restrições de tempo e custo (VARGAS, 2009).

Temporário significa que por maior que seja o tempo de seu desenvolvimento, ele terá um fim. Um projeto não dura para sempre. E terá fim assim que seus objetivos forem alcançados ou quando se chegar à conclusão de que estes objetivos não serão ou não poderão mais ser atingidos. Ou também, porque a necessidade do projeto não existe mais. Embora um projeto seja temporário, seu resultado tende a ser duradouro ou permanente, podendo ser classificado também como resultado tangível ou intangível (TRENTIM, 2011), conforme exemplos:

- Produção de um produto, item final ou componente.
- Documentos, criação de conhecimento, relatórios de auditorias, análise de mercados ou processos.
- Desenvolvimento ou aquisição de um sistema (novo ou modificado).
- Construção de infraestrutura, implementação de novos procedimentos ou processos de negócios.

Um projeto é **único**, pois mesmo que já tenha sido executado em outras circunstâncias ou organizações, ao ser colocado em prática novamente, o cenário envolvido (necessidades, pessoas, tecnologias) já mudou. A realidade do projeto atual já é outra em decorrência das mudanças ocorridas.

É considerado como **progressivo,** pois à medida que temos mais conhecimento sobre ele, vamos elaborando-o progressivamente, melhorando o seu detalhamento e as particularidades que o definem como único. Mas, podemos dizer que a característica de destaque de um projeto é o **risco**, pois nunca podemos ter a certeza de que o mesmo será bem-sucedido.

Em contraste com os projetos, encontram-se as operações de uma organização. Estas são esforços de trabalho contínuo, que seguem padrões organizacionais em sua execução, têm caráter repetitivo e rotineiro, e são acima de tudo, permanentes (TRENTIM, 2011).

Principais diferenças entre projetos e operações:

- Projetos têm um começo e fim definidos. Produzem um resultado ou produto único.
- Operações são repetitivas e produzem os mesmos resultados cada vez que o processo é executado.

Os projetos para serem lucrativos, executados no prazo e entregues com a devida qualidade, precisam de **gerenciamento**, cujo objetivo consiste em planejar e controlar atividades sincronizadas de diversas pessoas.

Segundo o PMI (2000), podemos entender gerenciamento de projetos como: "a conversão de uma necessidade, especificada por objetivos e critérios de sucesso, em um resultado alcançado".

Cada projeto possui um ciclo de vida, que ajuda a definir o início e término de cada etapa, o que deve ser realizado e por quem deve ser executado (matriz de responsabilidade do projeto). Serve para dar alicerce ao tripé de sucesso dos projetos: tempo/custo/qualidade. A entrega deve ser feita no prazo estipulado, dentro do orçamento apontado, com nível de qualidade atendendo as necessidades do cliente comprador (VARGAS, 2009).

Devido à dinamicidade e competitividade do mercado, diversas empresas têm buscado alternativas tecnológicas para otimizar seus processos, diminuir custos e maximizar lucros, melhorar a qualidade de seus serviços e produtos, com o objetivo de atingir ou manter-se em um patamar importante no mercado em que atuam.

Com a grande demanda do mercado por soluções de *software*, empresas de desenvolvimento de *software* necessitam desenvolver diversos projetos simultaneamente, onde cada um possui seus objetivos específicos. Esses projetos precisam ser gerenciados estrategicamente de forma a se alcançar os objetivos individuais dos projetos e manterem-se alinhados com as metas da organização.

Projetos de desenvolvimento de *software* estão vulneráveis a eventos que podem afetar negativamente o cronograma, custo, escopo do projeto, e a qualidade do produto final. Em um ambiente onde há múltiplos projetos, além de impactar nos objetivos de cada projeto, eventos adversos podem ameaçar o sucesso das metas organizacionais.

FONTE: Disponível em: <http://www.contecsi.fea.usp.br/envio/index.php/contecsi/9contecsi/paper/viewFile/3508/2007>. Acesso em: 22 fev. 2016.

O gerente de projetos é o responsável pelo controle e bom desempenho dos projetos. Deve ser aceito pela equipe. Não precisa ter domínio técnico, uma vez que suas habilidades devem estar mais voltadas para a área de gestão (DINSMORE; CAVALIERI, 2003). Ainda de acordo com os autores, o gerente de projetos deve ser um facilitador, não tendo falhas na liderança, nos seus processos comunicacionais, nas formas de negociação adotadas, e na condução da resolução de problemas.

2 PROJETO ORIENTADO A OBJETOS

Os projetos fundamentados e desenvolvidos conceitualmente em Orientação a Objetos propõem analisar, estruturar e interligar as partes de acordo com as funções específicas. O alicerce são os objetos cujo desenvolvimento é independente e que em uma fase adiantada do projeto, se relacionam/interligam dando origem ao projeto em sua totalidade.

Podemos considerar três etapas macro, no desenvolvimento de um projeto Orientado a Objetos:

- A Análise Orientada a Objeto;
- O Projeto Orientado a Objeto;
- A Programação Orientada a Objetos.

Quando pensamos no bom andamento do desenvolvimento do projeto, devemos ter em mente que estas etapas são sequenciais e dependentes, ou seja, para iniciar uma nova etapa, a anterior teoricamente deverá estar finalizada.

3 CARACTERÍSTICAS DOS PROJETOS ORIENTADOS A OBJETOS

Projeto orientado a objeto é parte do desenvolvimento orientado a objeto.

- Projeto orientado a objetos é um formato de desenvolvimento, onde os envolvidos no desenvolvimento pensam nas coisas ao invés de focar em suas funções.
- A arquitetura é projetada pensando nos serviços oferecidos pelos objetos.
- Objetos são abstrações do mundo real ou entidades do sistema e são autogerenciáveis.
- Objetos são independentes entre si e encapsulam representações de informação e estado.
- As áreas de dados compartilhados são eliminadas.
- A comunicação dos objetos ocorre através do processo de envio e recebimento de mensagens.

Divide-se em três fases:

- Análise OO – propõe o desenvolvimento de um modelo OO focado no domínio do aplicativo a ser desenvolvido. A identificação dos objetos relaciona-se com a realidade da situação problema a ser resolvida.
- Projeto OO – propõe a criação de um modelo OO para implementar os requisitos do problema que deverá ser resolvido.
- Programação OO – é a execução das etapas acima, implementada por uma linguagem de programação OO.

3.1 PROGRAMAÇÃO ORIENTADA A OBJETOS

3.1.1 Histórico

Motivada pela necessidade de resolver os problemas da crise do *software* na década de 60, surgiu a Engenharia do *Software*, porém, algumas técnicas desenvolvidas entre 1970 e 1980 não foram suficientes para acabar com os problemas de produtividade e qualidade no desenvolvimento de *software* da época.

A evolução tecnológica e o surgimento da internet na década de 90 fizeram com que grande parte de empresas de médio e grande porte já suportassem suas informações em um considerável número de sistemas de informação. Nesta época, o computador tornou-se uma indispensável ferramenta de trabalho.

Esse aumento na utilização de sistemas informatizados expõe a necessidade de desenvolvimento de *softwares* mais robustos, mais dinâmicos e com troca de informação mais segura. Uma das características atuais no desenvolvimento de aplicativos, é a interação dos mesmos com os usuários. Isso só foi possível após o surgimento das interfaces gráficas, cujo objetivo é intermediar a base de dados e o usuário final. Outras características são a manutenibilidade, portabilidade e a interação com outros aplicativos, favorecendo a rápida troca e atualização de informações (MELO, 2006).

Mesmo assim, produtividade e qualidade continuavam sendo um problema não resolvido. Após um período de estudos e testes, engenheiros de *software* perceberam a necessidade da reutilização de códigos e componentes.

3.1.2 Fundamentos da Programação Orientada a Objetos

Orientação a objetos pode ser considerada uma tecnologia que define os sistemas como uma coleção de objetos e suas funcionalidades. Esta tecnologia permitiu colocar em prática o conceito de reusabilidade no desenvolvimento de *software*. Conforme você deve lembrar, é pautada nos princípios de abstração, hierarquização, encapsulamento, classificação, modularização, relacionamento, simultaneidade e persistência (TACLA, 2010).

4 O PARADIGMA DA ORIENTAÇÃO A OBJETOS

Representar fielmente as situações do mundo real, é a proposta da Orientação a Objetos. Por este motivo, o conceito não vê um sistema computacional como um conjunto de processos, mas como uma coleção de objetos que interagem entre si.

Logo, os sistemas OO têm uma estrutura diferente. São disponibilizados em módulos que contêm estado e suas operações, e permitem a reutilização do código, através da herança. Polimorfismo também é importante, uma vez que permite escolher funcionalidades que um determinado programa poderá assumir assim que for executado.

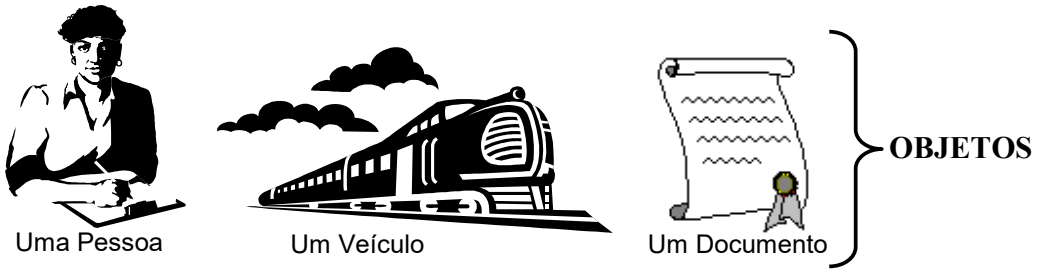
A Linguagem Simula 67, foi a primeira a possibilitar a implementação do conceito de objeto. Em seguida, os conceitos foram refinados na linguagem Smalltalk 80, que ainda hoje auxilia sendo protótipo de implementação dos modelos orientados a objetos. A linguagem mais popular hoje em dia é Java, porém, muitos adeptos consideram que a Smalltalk 80 é a única linguagem de programação integralmente orientada a objetos (VIDEIRA, 2008)

Revisando, o modelo orientado a objetos é formado por quatro itens básicos:

"Objeto: é uma abstração encapsulada que tem um estado interno dado por uma lista de atributos cujos valores são únicos para o objeto. O objeto também conhece uma lista de mensagens que ele pode responder e sabe como responder cada uma". (BORGES; CLINIO, 2015, p. 13).

Para ilustrar, pode-se dizer que objetos representam uma coleção de dados relacionados com um tema em comum, como exemplificado na figura a seguir.

FIGURA 1 – EXEMPLO DE OBJETO

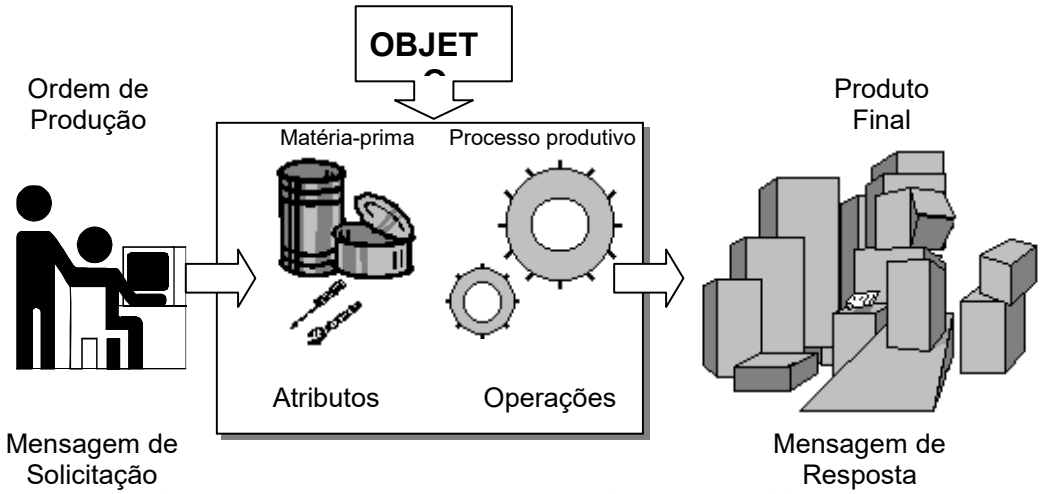


FONTE: Disponível em: <http://www.inf.pucrs.br/gustavo/disciplinas/pli/material/paradigmas-aula12.pdf>. Acesso em: 1 dez. 2015

- **Mensagem**: mensagens são solicitações de um objeto para outro, para que o objeto receptor produza um resultado.

A figura a seguir ilustra o funcionamento de envio/recebimento das mensagens e interação entre os objetos.

FIGURA 2 – ENVIO E RECEBIMENTO DE MENSAGENS



FONTE: Disponível em: <http://www.inf.pucrs.br/gustavo/disciplinas/pli/material/paradigmas-aula12.pdf>. Acesso em: 1 dez. 2015.

- **Métodos**: método pode ser entendido como um procedimento que o objeto executa quando recebe uma mensagem.

- **Atributos**: são as características do objeto.

- **Classes**: guarda um conjunto de atributos que define as características de um objeto.

- **Encapsulamento:** é a visão do objeto em seu estado interno, com mensagens e métodos.

- **Polimorfismo**: é a propriedade que permite que a mesma mensagem seja enviada para mais de um objeto.

- **Herança**: é uma das propriedades mais importantes do modelo de orientação a objetos. As subclasses herdam todos os componentes da classe pai. A herança possibilita uma definição de novas classes, sem duplicação de código.

O quadro a seguir resume os conceitos.

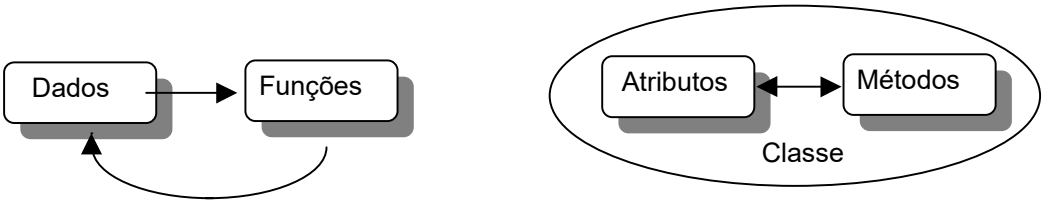
QUADRO 1 – RESUMO DOS CONCEITOS DE ORIENTAÇÃO A OBJETOS

Palavra-Chave	Breve Definição	Exemplo
Classe	Agrupamento de objetos similares que apresentam os mesmos atributos e operações	Indivíduo, caracterizando as pessoas do mundo
Atributo	Característica particular de uma ocorrência da classe	Indivíduo possui nome, sexo, data de nascimento
Operações	Lógica contida em uma classe para designar-lhe um comportamento	Cálculo da idade de uma pessoa em uma classe (Indivíduo)
Encapsulamento	Combinação de atributos e operações de uma classe	Atributo: data de nascimento Operação: cálculo da idade
Herança	Compartilhamento pela subclasse dos atributos e operações da classe pai	Subclasse (Eucalipto) compartilha atributos e operações da classe (Árvore)
Subclasse	Característica particular de uma classe	Classe (Árvore) → Subclasses (Ipê, Eucalipto, Jacarandá, etc.)
Instância de Classe	Uma ocorrência específica de uma classe. É o mesmo que objeto	Uma pessoa, uma organização ou um equipamento
Objeto	Elemento do mundo real (natureza). Sinônimo de instância de classe	Pessoa "Fulano de Tal", Organização "ACM", Equipamento "Extintor"
Mensagem	Uma solicitação entre objetos para invocar certa operação	Informar idade da pessoa "Fulano de Tal"
Polimorfismo	Habilidade para usar a mesma mensagem para invocar comportamentos diferentes do objeto	Chamada da operação: "Calcular Saldo" de correntista. Invoca as derivações correspondentes para cálculo de saldo de poupança, renda fixa, etc.

FONTE: Disponível em: <http://www.inf.pucrs.br/gustavo/disciplinas/pli/material/paradigmas-aula12.pdf>. Acesso em: 1 dez. 2015.

O encapsulamento é uma parte importante pois impede que algumas características dos objetos de uma classe possam ser vistas ou modificadas externamente. Restringe a visibilidade do objeto, mas permite o reuso. A figura a seguir representa a situação.

FIGURA 3 – PROGRAMAÇÃO CONVENCIONAL X PROGRAMAÇÃO BASEADA EM OBJETOS



FONTE: <http://www.inf.pucrs.br/gustavo/disciplinas/pli/material/paradigmas-aula12.pdf>. Acesso em: 1 dez. 2015.

Porém, estas duas características podem ser incompatíveis (visibilidade e reuso). Por exemplo: em determinada situação, ao se adicionar operações torna-se necessário o acesso ao detalhamento interno da implementação!

Em resumo, conforme o exposto anteriormente e em cadernos de disciplinas correspondentes, você deve ter percebido que a modelagem para a construção de sistemas OO, estrutura o problema a ser resolvido em forma de objetos do mundo real que interagem simultaneamente, tendo suas características representadas através de atributos (dados) e operações (processos).

a. Vantagens do Projeto OO

Conforme Melo (2006), as vantagens de um projeto orientando a objetos são:

- Facilidade de manutenção: torna mais rápidas as atividades de programação e manutenção de sistemas de informação.
- Os modelos gerados permitem maior compreensão.
- A reutilização do código é mais eficiente.
- A comunicação é mais segura e rápida.
- Percebe-se uma maior estabilidade e flexibilidade nas construções.

b. Processo de análise OO

Para Larman (2007), o processo de análise exige obrigatoriamente a execução e cumprimento dos passos abaixo:

- Identificação do problema a ser resolvido.
- Definição dos casos de uso.
- Identificação dos objetos envolvidos na situação problema.
- Desenvolvimento do modelo conceitual através dos diagramas de classe e relacionamento.
- Especificação dos diagramas de sequência, atividade, objeto e demais diagramas necessários para a composição do modelo de solução.

Modelos são usados como forma de representação em todas as situações onde se faz necessária a abstração para entender determinadas situações ou problemas. Vários são os objetivos dos modelos:

- Permitem uma melhor compreensão de situações reais.
- Validam situações antes das mesmas tomarem forma física.
- Servem como ferramenta de comunicação para demonstrar como algo está sendo desenvolvido/construído ou simplesmente para demonstrar ideias.
- Reduzem a complexidade das situações.

De acordo com Tacla (2010), no desenvolvimento de projetos de *software* especificamente, é na fase de análise do projeto que os modelos alcançam o auge de sua importância, pois permitem que uma situação seja entendida e assimilada antes mesmo de sua solução ser implementada. Um bom modelo se concentra em resolver um problema principal, não dando tanta importância às situações secundárias. Um modelo nasce, portanto, com o objetivo de resolver uma situação problema específica.

4.1 MODELOS NA ANÁLISE ORIENTADA A OBJETOS

A modelagem é a base da tecnologia de orientação a objetos, tendo como base duas dimensões principais:

- A **dimensão estrutural dos objetos**, que inclui a identidade de cada objeto, sua classificação, seu encapsulamento e seus relacionamentos;
- A **dimensão dinâmica do comportamento dos objetos**, que inclui a definição dos estados válidos dos objetos e a especificação dos eventos que causam a transição desses estados.

Algumas literaturas apresentam uma terceira opção, que se refere à dimensão funcional dos requisitos, referenciando as funções de transformação do sistema (MELO, 2006).

Para elencar e documentar todos os detalhes e características do sistema torna-se necessário optar por um método para especificar a situação problema, através de uma notação expressiva e bem definida. A padronização da notação possibilita aos analistas, projetistas e desenvolvedores documentar e descrever o cenário de trabalho fácil de entender e de ser transmitido aos demais envolvidos no projeto.

Para isso, alguns aspectos devem ser considerados na escolha do modelo, conforme aponta (MELO, 2006):

- O modelo deve ser bem documentado através de livros ou artigos que facilitem o seu entendimento.

- A equipe de desenvolvimento deve estar familiarizada com o modelo.
- Ele deve concentrar-se nos princípios da orientação a objetos.
- Deve ser conciso e enxuto o suficiente para captar os aspectos relevantes do sistema.
- A melhor metodologia é, provavelmente, aquela que você já sabe usar.

Os principais modelos ou métodos de análise orientada a objetos, são propostos por:

- Booch
- Rumbaugh
- Jacobson
- Coad e Yourdon
- Wirfs-Brock

4.1.1 Modelo de *software*

A modelagem de um *software* implica em criar modelos de *software*, mas o que é realmente um modelo de *software*? Um modelo de *software* captura uma visão de um sistema físico, é uma abstração do sistema com um certo propósito, como descrever aspectos estruturais ou comportamentais do *software*. Esse propósito determina o que deve ser incluído no modelo e o que é considerado irrelevante. Assim um modelo descreve completamente aqueles aspectos do sistema físico que são relevantes ao propósito do modelo, no nível apropriado de detalhe. Dessa forma, um modelo de casos de uso fornecerá uma visão dos requisitos necessários ao sistema, identificando as funcionalidades do *software* e os atores que poderão utilizá-las, não se preocupando em detalhar nada além disso. Já um modelo conceitual irá identificar as classes relacionadas ao domínio do problema, sem detalhar seus métodos, enquanto um modelo de domínio ampliará o modelo conceitual, incluindo informações relativas à solução do problema, incluindo, entre outras coisas, os métodos necessários a essa solução.

4.1.2 Levantamento e Análise de Requisitos

Uma das primeiras fases de um processo de desenvolvimento de *software* consiste no Levantamento de Requisitos. As outras etapas, sugeridas por muitos autores, são: Análise de Requisitos, Projeto, que se constitui na principal fase da modelagem, Codificação, Testes e Implantação.

Dependendo do método/processo adotado, essas etapas ganham, por vezes, nomenclaturas diferentes, podendo algumas delas ser condensa-

das em uma etapa única, ou uma etapa pode ser dividida em duas ou mais etapas. Se tomarmos como exemplo o Processo Unificado (*Unified Process*), um método de desenvolvimento de *software*, veremos que este se divide em quatro fases: Concepção, onde é feito o levantamento de requisitos; Elaboração, onde é feita a análise dos requisitos e o projeto do *software*; Construção, onde o *software* é implementado e testado; e Transição, onde o *software* será implantado. As fases de Elaboração e Construção ocorrem, sempre que possível, em ciclos iterativos, dessa forma, sempre que um ciclo é completado pela fase de Construção, volta-se à fase de Elaboração para tratar do ciclo seguinte, até todo o *software* ser finalizado.

As etapas de levantamento e análise de requisitos trabalham com o domínio do problema e tentam determinar "o que" o *software* deve fazer e se é realmente possível desenvolver o *software* solicitado. Na etapa de levantamento de requisitos, o engenheiro de *software* busca compreender as necessidades do usuário e o que ele deseja que o sistema a ser desenvolvido realize. Isso é feito sobretudo por meio de entrevistas, nas quais o engenheiro tenta compreender como funciona atualmente o processo a ser informatizado e quais serviços o cliente precisa que o *software* forneça.

Devem ser realizadas tantas entrevistas quantas forem necessárias para que as necessidades do usuário sejam bem compreendidas. Durante as entrevistas, o engenheiro deve auxiliar o cliente a definir quais informações deverão ser produzidas, quais deverão ser fornecidas e qual o nível de desempenho exigido do *software*. Um dos principais problemas enfrentados na fase de levantamento de requisitos é o de comunicação. A comunicação constitui-se em um dos maiores desafios da engenharia de *software*, caracterizando-se pela dificuldade em conseguir compreender um conjunto de conceitos vagos, abstratos e difusos que representam as necessidades e os desejos dos clientes e transformá-los em conceitos concretos e inteligíveis. A fase de levantamento de requisitos deve identificar dois tipos de requisitos: os funcionais e os não-funcionais.

Os requisitos funcionais correspondem ao que o cliente quer que o sistema realize, ou seja, as funcionalidades do *software*. Já os requisitos não-funcionais correspondem às restrições, condições, consistências, validações que devem ser levadas a efeito sobre os requisitos funcionais. Por exemplo, em um sistema bancário deve ser oferecida a opção de abrir novas contas correntes, o que é um requisito funcional. Já determinar que somente pessoas maiores de idade possam abrir contas corrente é um requisito não-funcional.

Podem existir diversos tipos de requisitos não-funcionais, como de usabilidade, desempenho, confiabilidade, segurança ou interface. Alguns requisitos não-funcionais identificam regras de negócio, ou seja, as polí-

ticas, normas e condições estabelecidas pela empresa que devem ser seguidas na execução de uma funcionalidade. Por exemplo, estabelecer que depois de abrir uma conta é necessário depositar um valor mínimo inicial é uma regra de negócio adotada por um determinado banco e que não necessariamente é seguida por outras instituições bancárias. Outro exemplo de regra de negócio seria determinar que, em um sistema de videolocadora, só se poderia realizar uma nova locação para um sócio depois de ele ter devolvido as cópias locadas anteriormente. Logo após o levantamento de requisitos, passa-se à fase em que as necessidades apresentadas pelo cliente são analisadas.

Essa etapa é conhecida como análise de requisitos. Aqui o engenheiro examina os requisitos enunciados pelos usuários, verificando se estes foram especificados corretamente e se foram realmente bem compreendidos. A partir da etapa de análise de requisitos são determinadas as reais necessidades do sistema de informação. A grande questão é: como saber se as necessidades dos usuários foram realmente bem compreendidas?

Um dos objetivos da análise de requisitos consiste em determinar se as necessidades dos usuários foram entendidas de maneira correta, verificando se alguma questão deixou de ser abordada, determinando se algum requisito foi especificado incorretamente ou se algum conceito precisa ser melhor explicado. Durante a análise de requisitos, uma linguagem de modelagem auxilia a levantar questões que não foram concebidas durante as entrevistas iniciais.

Tais questões devem ser sanadas o quanto antes, para que o projeto do *software* não tenha que sofrer modificações quando seu desenvolvimento já estiver em andamento, o que pode causar significativos atrasos no desenvolvimento do *software*, sendo por vezes necessário remodelar por completo o projeto. Além daquele concernente à comunicação, outro grande problema encontrado durante as entrevistas consiste no fato de que, na maioria das vezes, os usuários não têm realmente certeza do que querem e não conseguem enxergar as reais potencialidades de um sistema de informação.

Em geral, os engenheiros de *software* precisam sugerir inúmeras características e funções do sistema que o cliente não sabia como formular ou sequer havia imaginado. Na realidade, na maior parte das vezes, esses profissionais precisam reestruturar o modo como as informações são geridas e utilizadas pela empresa e apresentar maneiras de combiná-las e apresentá-las de maneira que possam ser melhor aproveitadas pelos usuários. Em muitos casos é realmente isso o que os clientes esperam dos engenheiros de *software*, porém, em outros, os engenheiros encontram fortes resistências a qualquer mudança na forma como a empresa manipula suas informações, fazendo-se necessário um significativo esforço para provar ao cliente que

as modificações sugeridas permitirão um melhor desempenho do *software*, além de ser útil para a própria empresa, obviamente.

Na realidade, nesse último caso é fundamental trabalhar bastante o aspecto social da implantação de um sistema informatizado na empresa, pois muitas vezes a resistência não é tanto por parte da gerência, mas pelos usuários finais, que serão obrigados a mudar a forma como estavam acostumados a trabalhar e aprender a utilizar uma nova tecnologia.

4.1.3 Prototipação

A prototipação é uma técnica bastante popular e de fácil aplicação. Essa técnica consiste em desenvolver rapidamente um "rascunho" do que seria o sistema de informação quando ele estivesse finalizado. Um protótipo normalmente apresenta pouco mais do que a interface do *software* a ser desenvolvido, ilustrando como as informações seriam inseridas e recuperadas no sistema, apresentando alguns exemplos com dados fictícios de quais seriam os resultados apresentados pelo *software*, principalmente em forma de relatórios. A utilização de um protótipo pode, assim, evitar que, após meses ou até anos de desenvolvimento, descubra-se, ao implantar o sistema, que o *software* não atende completamente às necessidades do cliente, devido, sobretudo, a falhas de comunicação durante as entrevistas iniciais.

Hoje em dia, é possível desenvolver protótipos com extrema rapidez e facilidade, por meio da utilização de ferramentas conhecidas como RAD (*Rapid Application Development* ou Desenvolvimento Rápido de Aplicações). Essas ferramentas são encontradas na maioria dos ambientes de desenvolvimento das linguagens de programação atuais, como NetBeans, Delphi, Visual Basic ou C++ Builder, entre outras. Essas ferramentas disponibilizam ambientes de desenvolvimento que permitem a construção de interfaces de forma muito rápida, além de permitirem também modificar tais interfaces de maneira igualmente veloz, na maioria das vezes sem a necessidade de alterar qualquer código porventura já escrito.

As ferramentas RAD permitem a criação de formulários e a inserção de componentes nos mesmos, de uma forma muito simples, rápida e fácil, bastando ao desenvolvedor selecionar o componente (botões, caixas de texto, *labels*, combos etc.) em uma barra de ferramentas e clicar com o *mouse* sobre o formulário. Alternativamente, o usuário pode clicar duas vezes sobre o componente desejado, fazendo com que um componente do tipo selecionado surja no centro do formulário. Além disso, tais ferramentas permitem ao usuário mudar a posição dos componentes depois de terem sido colocados no formulário simplesmente selecionando o componente com o *mouse* e o arrastando para a posição desejada.

Esse tipo de ferramenta é extremamente útil no desenvolvimento de protótipos pela facilidade de produzir e modificar as interfaces. Assim, depois de determinar quais as modificações necessárias ao sistema de informação após o protótipo ter sido apresentado aos usuários, pode-se modificar a interface do protótipo de acordo com as novas especificações e reapresentá-lo ao cliente de forma muito rápida. Seguindo esse raciocínio, a etapa de análise de requisitos deve, obrigatoriamente, produzir um protótipo para demonstrar como se apresentará e comportará o sistema em essência, bem como quais informações deverão ser inseridas no sistema e que tipo de informações deverão ser fornecidas pelo *software*. Um protótipo é de extrema importância durante as primeiras fases de engenharia de um sistema de informação.

Por meio da ilustração que um protótipo pode apresentar, a maioria das dúvidas e erros de especificação pode ser sanada, devido ao fato de um protótipo demonstrar visualmente um exemplo de como funcionará o sistema depois de concluído, como será sua interface, de que maneira os usuários interagirão com ele, que tipo de relatórios serão fornecidos etc., facilitando a compreensão do cliente. Apesar das grandes vantagens advindas do uso da técnica de prototipação, é necessária ainda uma ressalva: um protótipo pode induzir o cliente a acreditar que o *software* encontra-se em um estágio bastante avançado de desenvolvimento.

Com frequência ocorre de o cliente não compreender o conceito de um protótipo. Para ele, o esboço apresentado já é o próprio sistema praticamente acabado. Por isso, muitas vezes o cliente não compreende nem aceita prazos longos, os quais considera absurdos, já que o sistema lhe foi apresentado já funcionando, necessitando de alguns poucos ajustes. Por isso, é preciso deixar bem claro ao usuário que o *software* que lhe está sendo apresentado é apenas um "rascunho" do que será o sistema de informação quando estiver finalizado e que seu desenvolvimento ainda não foi realmente iniciado.

4.1.4 Prazos e Custos

Em seguida, vem a espinhosa e desagradável, porém extremamente importante, questão dos prazos e custos. Como determinar o prazo real de entrega de um *software*? Quantos profissionais deverão trabalhar no projeto? Qual será o custo total de desenvolvimento? Qual deverá ser o valor estipulado para produzir o sistema? Como determinar a real complexidade de desenvolvimento do *software*? Geralmente, após as primeiras entrevistas, os clientes estão bastante interessados em saber quanto vai lhes custar o sistema de informação e em quanto tempo eles o terão implantado e funcionando em sua empresa.

A estimativa de tempo é realmente um tópico extremamente complexo da engenharia de *software*. Na realidade, por melhor modelado que um sistema tenha sido, ainda assim fica difícil determinar com exatidão os prazos finais de entrega do *software*. Uma boa modelagem auxilia a estimar a complexidade de desenvolvimento de um sistema, e isso, por sua vez, ajuda – e muito – a determinar o prazo final em que o *software* será entregue. No entanto, é preciso ter diversos sistemas de informação com níveis de dificuldade e características semelhantes ao *software* que está para ser construído, já previamente desenvolvidos e bem documentados, para determinar com maior exatidão a estimativa de prazos.

Contudo, mesmo com o auxílio dessa documentação, ainda é muito difícil estipular uma data exata. O máximo que se pode conseguir é apresentar uma que seja aproximada, com base na experiência documentada de desenvolvimento de outros *softwares*. Assim, é recomendável acrescentar alguns meses à data de entrega, o que serve como margem de segurança para possíveis erros de estimativa. Para poder auxiliar na estimativa de prazos e custos de um *software*, a documentação da empresa desenvolvedora deverá ter registros das datas de início e término de cada projeto já concluído, além do custo real de desenvolvimento que tais projetos acarretaram, envolvendo inclusive os custos com manutenção e o número de profissionais envolvidos em cada projeto.

Na verdade, uma empresa de desenvolvimento de *software* que nunca tenha desenvolvido um sistema de informação antes e, portanto, não tenha documentação histórica de projetos anteriores, dificilmente será capaz de apresentar uma estimativa correta de prazos e custos, principalmente porque a equipe de desenvolvimento não saberá com certeza quanto tempo levará desenvolvendo o sistema, já que o tempo de desenvolvimento influencia diretamente o custo de desenvolvimento do sistema e, logicamente, o valor a ser cobrado pelo *software*.

Se a estimativa de prazo estiver errada, cada dia a mais de desenvolvimento do projeto acarretará prejuízos para a empresa que desenvolve o sistema, decorrentes, por exemplo, de pagamentos de salários aos profissionais envolvidos no projeto que não haviam sido previstos e desgaste dos equipamentos utilizados. Isso sem levar em conta prejuízos mais difíceis de contabilizar, como manter inúmeros profissionais ocupados em projetos que já deveriam estar concluídos, que deixam de trabalhar em novos projetos, além da insatisfação dos clientes por não receberem o produto no prazo estimado e a propaganda negativa daí decorrente.

4.1.5 Projeto

Enquanto a fase de análise trabalha com o domínio do problema, a fase de projeto trabalha com o domínio da solução, procurando estabelecer "como" o sistema fará o que foi determinado na fase de análise, ou seja,

qual será a solução para o problema identificado. É na etapa de projeto que é realizada a maior parte da modelagem do *software* a ser desenvolvido, ou seja, é nessa etapa que é produzida a arquitetura do sistema.

A etapa de projeto toma a modelagem iniciada na fase de análise e lhe acrescenta profundos acréscimos e detalhamentos. Enquanto na análise foram identificadas as funcionalidades necessárias ao *software* e suas restrições, na fase de projeto será estabelecido como essas funcionalidades deverão realizar o que foi solicitado. A fase de projeto leva em consideração os recursos tecnológicos existentes para que o problema apresentado pelo cliente possa ser solucionado. É nesse momento que será selecionada a linguagem de programação a ser utilizada, o sistema gerenciador de banco de dados a ser empregado, como será a interface final do sistema e até mesmo como o *software* será distribuído fisicamente na empresa, especificando o *hardware* necessário para a sua implantação e funcionamento correto.

4.1.6 Manutenção

Possivelmente a questão mais importante que todas as outras já enunciadas é a da manutenção. Alguns autores afirmam que muitas vezes a manutenção de um *software* pode representar de 40 a 60% do custo total do projeto. Alguém poderá então dizer que a modelagem é necessária para diminuir os custos com manutenção – se a modelagem estiver correta o sistema não apresentará erros e, então, não precisará sofrer manutenções. Embora um dos objetivos de modelar um *software* seja realmente diminuir a necessidade de mantê-lo, a modelagem não serve apenas para isso.

Na maioria dos casos, a manutenção de um software é inevitável, pois, como já foi dito, as necessidades de uma empresa são dinâmicas e mudam constantemente, o que faz surgir novas necessidades que não existiam na época em que o *software* foi projetado, isso sem falar nas frequentes mudanças em leis, alíquotas, impostos, taxas ou formato de notas fiscais, por exemplo. Levando isso em consideração, é bastante provável que um sistema de informação, por mais bem modelado que esteja, precise sofrer manutenções.

Nesse caso, a modelagem não serve apenas para diminuir a necessidade de futuras manutenções, mas também para facilitar a compreensão do sistema por quem tiver que mantê-lo, já que, em geral, a manutenção de um sistema é considerada uma tarefa ingrata pelos profissionais de desenvolvimento, por normalmente exigir que estes despendam grandes esforços para compreender códigos escritos por outros cujos estilos de desenvolvimento são diferentes e que, via de regra, não se encontram mais na empresa. Esse tipo de código é conhecido como "código alienígena" ou "*software* legado".

O termo refere-se a códigos que não seguem as regras atuais de desenvolvimento da empresa, não foram modelados e, por conseguinte, têm pouca ou nenhuma documentação. Além disso, nenhum dos profissionais da equipe atual trabalhou em seu projeto inicial e, para piorar, o código já sofreu manutenções anteriores por outros profissionais que também não se encontram mais na empresa, sendo que cada um deles tinha um estilo de desenvolvimento diferente, ou seja, como se diz no meio de desenvolvimento, o código encontra-se "remendado". Assim, uma modelagem correta aliada a uma documentação completa e atualizada de um sistema de informação torna mais rápido o processo de manutenção e impede que erros sejam cometidos, já que é muito comum que, depois de manter uma rotina ou função de um *software*, outras rotinas ou funções do sistema que antes funcionavam perfeitamente passem a apresentar erros ou simplesmente deixem de funcionar.

Tais erros são conhecidos como "efeitos colaterais" da manutenção. Além disso, qualquer manutenção a ser realizada em um sistema deve ser também modelada e documentada, para não desatualizar a documentação do sistema e prejudicar futuras manutenções, já que muitas vezes uma documentação desatualizada pode ser mais prejudicial à manutenção do sistema do que nenhuma documentação. Pode-se fornecer uma analogia de "manutenção" na vida real responsável pela produção de um efeito colateral no meio ambiente, o que não deixa de ser um sistema: muito pelo contrário, é "o" sistema.

Na realidade, esse exemplo não identifica exatamente uma manutenção, e sim uma modificação em uma região. Descobri recentemente, ao assistir a uma reportagem, que a formiga saúva vinha se tornando uma praga em algumas regiões do país porque estava se reproduzindo demais. Esse crescimento desordenado era causado pelos tratores que, ao arar a terra, destruíam os formigueiros da formiga lava-Pés, que ficam próximos à superfície, mas não afetavam os formigueiros de saúvas, por estes se encontrarem em um nível mais profundo do solo. Entretanto, as lava-pés consumiam os ovos das saúvas, o que impedia que estas aumentassem demais. Assim, a diminuição das formigas lava-pés resultou no crescimento desordenado das saúvas. Isso é um exemplo de manutenção com efeito colateral na vida real. No caso, foi aplicada uma espécie de "manutenção", onde modificou-se o ambiente para arar a terra e produziu-se uma praga que antes constituía-se apenas em uma espécie controlada.

Se a "função" da formiga lava-pés estivesse devidamente documentada, ela não teria sido eliminada, e a saúva, por conseguinte, não teria se tornado uma praga.

4.1.7 Documentação Histórica

Finalmente, existe a questão da perspectiva histórica, ou seja, novamente a já tão falada documentação de *software*. Neste caso, referimo-nos à documentação histórica dos projetos anteriores já concluídos pela empresa.

É por meio dessa documentação histórica que a empresa pode responder a perguntas como:

- A empresa está evoluindo?
- O processo de desenvolvimento tornou-se mais rápido?
- As metodologias hoje adotadas são superiores às práticas aplicadas anteriormente?
- A qualidade do *software* produzido está melhorando? Uma empresa ou setor de desenvolvimento de *software* necessita de um registro detalhado de cada um de seus sistemas de informação antes desenvolvidos para poder determinar, entre outros, fatores como:
    - A média de manutenções que um sistema sofre normalmente dentro de um determinado período de tempo.
    - Qual a média de custo de modelagem.
    - Qual a média de custo de desenvolvimento.
    - Qual a média de tempo despendido até a finalização do projeto.
    - Quantos profissionais são necessários envolver normalmente em um projeto.

Essas informações são computadas nos orçamentos de desenvolvimento de novos *softwares* e são de grande auxílio no momento de determinar prazos e custos mais próximos da realidade.

Além disso, a documentação pode ser muito útil em outra área: a Reusabilidade. Um dos grandes desejos e muitas vezes necessidades dos clientes é que o *software* esteja concluído o mais rápido possível.

Uma das formas de agilizar o processo de desenvolvimento é a reutilização de rotinas, funções e algoritmos previamente desenvolvidos em outros sistemas. Nesse caso, a documentação correta do sistema pode auxiliar a sanar questões como:

- Onde as rotinas se encontram?
- Para que foram utilizadas?
- Em que projetos estão documentadas?
- Elas são adequadas ao *software* atualmente em desenvolvimento?
- Qual o nível necessário de adaptação destas rotinas para que possam ser utilizadas na construção do sistema atual?

4.1.8 Por que tantos Diagramas?

Por que a UML é composta por tantos diagramas? O objetivo disso é fornecer múltiplas visões do sistema a ser modelado, analisando-o e modelando-o sob diversos aspectos, procurando-se, assim, atingir a completitude da modelagem, permitindo que cada diagrama complemente os outros. Cada diagrama da UML analisa o sistema, ou parte dele, sob uma determinada óptica. É como se o sistema fosse modelado em camadas, sendo que alguns diagramas enfocam o sistema de forma mais geral, apresentando uma visão externa do sistema, como é o objetivo do Diagrama de Casos de Uso, enquanto outros oferecem uma visão de uma camada mais profunda do *software*, apresentando um enfoque mais técnico ou ainda visualizando apenas uma característica específica do sistema ou um determinado processo. A utilização de diversos diagramas permite que falhas sejam descobertas, diminuindo a possibilidade da ocorrência de erros futuros.

FONTE: Extraído de: Guedes (2011, p. 20-30)

4.2 O PROBLEMA DA INTEGRAÇÃO DE MODELOS DE UMA METODOLOGIA

Muitos são os problemas que dificultam a eficácia dos modelos de análise, não sendo uma particularidade exclusiva da integração dos modelos. A consistência é tema de pauta de todas as metodologias.

Logo, construir modelos consistentes só é possível atualmente através da formalização, cuja real necessidade tem sido amplamente discutida por estudiosos da Orientação a Objetos. Neste sentido, alguns critérios são formulados por Larman (2007) para auxiliar a análise de metodologias:

- Existência de definição formal própria da metodologia para a sintaxe e para a semântica das técnicas usadas para modelar aspectos estáticos, dinâmicos e de interação entre objetos;
- Existência de definição do comportamento global do sistema em termos dos objetos individuais e da interação entre eles; e
- Existência de procedimento de verificação da consistência entre os modelos.

Apoiados nestes critérios, os autores Dedene e Snoeck (1994) analisaram as seguintes metodologias:

- AOO (OMT) de Rumbaugh et al.,
- FUSION de Hayes e Coleman,
- OSA de Embley et al.,
- OOSA de Shlaer e Mellor,
- OOA de Coad e Yourdon,
- O/B de Kappel e Schrefl,
- OOD de Booch e

- OOD de Wirfs-Brock et al., e concluíram que todas as abordagens fazem apenas o uso limitado de técnicas formais que praticamente não consideram a definição de sintaxe e de semântica do comportamento global do sistema e também da verificação de consistência entre modelos.

5 PROCESSO DE PROJETO ORIENTADO A OBJETOS

Após concluída a fase da análise orientada a objetos, dá-se início à fase do projeto propriamente dito, cujo planejamento e execução, obedece a algumas etapas, conforme os itens definidos abaixo:

- Compreender e definir o contexto e os modos de utilização do sistema.
- Projetar a arquitetura do sistema.
- Identificar os principais objetos do sistema.
- Descrever os modelos de projeto.
- Especificar as interfaces dos objetos.

Os passos acima são independentes, ou seja, para colocar em prática uma fase, não há a necessidade de a fase anterior estar concluída. Na maioria das vezes as fases são executadas de forma paralela, ocorrendo a execução de uma ou mais fases no mesmo período do projeto (MELO, 2006).

Veja uma demonstração de um projeto orientado a objetos em:

<https://www.youtube.com/watch?v=sRPeyxVR5XM>
<http://waprox.com/video-view/MKBgqs1VVss?utm_ref=related>
<https://www.youtube.com/watch?v=sF3TLkuy5T0>

É importante que você assista aos vídeos conforme a ordem indicada.

5.1 CONTEXTO DO SISTEMA E MODELOS DE USO

Para viabilizar um **Projeto de *Software* Orientado a Objeto** é primordialmente necessário definir como vai ocorrer a comunicação entre o usuário e o sistema, e as principais funcionalidades que o sistema disponibilizará para o mesmo. Somente a partir desta definição será possível estabelecer o relacionamento entre ambiente e sistema, conforme as possibilidades elencadas abaixo:

- A primeira situação vê o sistema como algo estático e descreve todos os subsistemas acoplados ao sistema principal, tendo um maior detalhamento das

partes envolvidas; oferece uma visão mais ampla do que realmente será ofertado ao usuário.

• A segunda situação se refere ao sistema como algo dinâmico, focando as formas de comunicação entre sistema e usuário, onde este tem maior domínio, podendo iniciar, desativar, relatar e testar o sistema, sendo que cada uma das operações fornecidas pelo sistema ao usuário desenvolve tarefas específicas que são realizadas por diversos subsistemas que estão presentes dentro do sistema principal (SOMMERVILLE, 2008)

5.2 PROJETO DA ARQUITETURA

Após você ter definido a forma de comunicação entre sistema e usuário, é hora de representar a arquitetura do sistema, através do mapeamento e formalização da camada de interface com o usuário, a camada de coleta de dados e a camada de instrumento que farão a captação das informações.

"A arquitetura em si deve ser desenvolvida tentando descrever o sistema da forma mais simples possível, não havendo mais do que sete entidades fundamentais relacionadas incluídas no modelo de arquitetura, ou seja, devemos procurar dividir o problema em no máximo sete camadas" (LARMAN, 2007, p. 56).

5.3 IDENTIFICAÇÃO DOS OBJETOS

Você vai perceber que conforme as fases do projeto vão sendo finalizadas, novas situações vão surgindo; novos problemas vão aparecendo, o que demanda a necessidade de criação de novos objetos e novas funcionalidades que serão incorporadas ao sistema como forma de corrigir ou evitar erros que possam acontecer futuramente quando o projeto estiver finalizado e em uso.

5.4 MODELOS DE OBJETOS

Modelos de objetos servem para especificar objetos e suas classes, e também, seus distintos relacionamentos. O modelo nada mais é do que o desenho do projeto em si. É o modelo que contém a informação entre o que é necessário no sistema, e como isso será alcançado. Por isso, os mesmos devem ser abstratos, não se importando com detalhes desnecessários ou secundários. O nível de detalhamento do projeto aliás, deve ir até o ponto de favorecer o claro entendimento dos programadores, para que os mesmos decidam de forma assertiva pelas melhores formas de implementação.

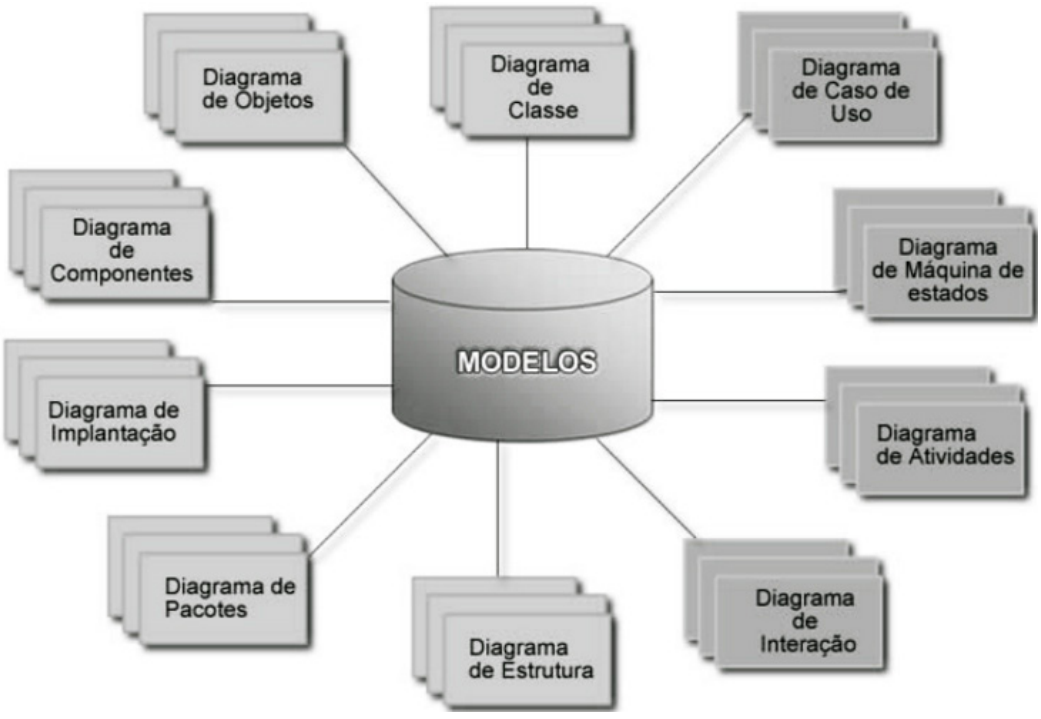
Dentre vários modelos, você terá que decidir por qual será o mais viável e lhe permitirá o maior nível de detalhamento para o seu projeto. Para isso, você deverá ter em mente que tipo de projeto/sistema está construindo, pois, cada sistema tem suas necessidades específicas e podem apontar para modelos distintos. Poucos sistemas utilizam-se de todos os modelos. Reduzir o número de modelos implica em economia de tempo, recursos financeiros e humanos no desenvolvimento.

Só para lembrar, existem dois tipos de modelos: estáticos e dinâmicos. Para a geração dos modelos utiliza-se atualmente a UML que se utiliza de diagramas na composição. Conforme você estudou no caderno de **Análise Orientada a Objetos II**, a versão 2.0 da UML traz consigo 13 diagramas, divididos em quatro grupos:

- diagramas estruturais;
- diagramas comportamentais;
- diagramas de interação;
- diagramas de implementação.

Relembre os 13 diagramas através da figura a seguir.

FIGURA 4 – DIAGRAMAS DA UML 2.0



FONTE: Piva (2010)

Para não esquecer, guarde o resumo dos diagramas exposto a seguir.

5.4.1 Diagramas Estruturais

- **De Classe**: este diagrama é fundamental e o mais utilizado na UML e serve de apoio aos outros diagramas. O Diagrama de Classe mostra o conjunto de classes com seus atributos e métodos e os relacionamentos entre classes.
- **De Objeto**: o Diagrama de Objeto está relacionado com o diagrama de classes e é praticamente um complemento dele. Fornece uma visão dos valores armazenados pelos objetos de um Diagrama de Classe em um determinado momento da execução do processo do *software*.
- **De Componentes**: está associado à linguagem de programação e tem por finalidade indicar os componentes do *software* e seus relacionamentos.
- **De Implantação**: determina as necessidades de *hardware* e características físicas do sistema.
- **De Pacotes**: representa os subsistemas englobados de forma a determinar partes que o compõem.
- **De Estrutura**: descreve a estrutura interna de um classificador.

FONTE: Disponível em: <http://issuu.com/geeadcps/docs/livro3_alta/81>. Acesso em: 7 out. 2015.

5.4.2 Diagramas Comportamentais

- **De Caso de Uso (*Use Case*)**: geral e informal para fases de levantamento e análise de Requisitos do Sistema.
- **De Máquina de Estados**: procura acompanhar as mudanças sofridas por um objeto dentro de um processo.
- **De Atividades**: descreve os passos a serem percorridos para a conclusão de uma atividade.
- **De Interação**: dividem-se em:
- **De Sequência**: descreve a ordem temporal em que as mensagens são trocadas entre os objetos.

- **Geral interação:** variação dos diagramas de atividades que fornece visão geral dentro do sistema ou processo do negócio.
- **Decomunicação**: associado ao diagrama de Sequência, complementando-o e concentrando-se em como os objetos estão vinculados.
- **De tempo**: descreve a mudança de estado ou condição de uma instância de uma classe ou seu papel durante o tempo.

FONTE: Disponível em: <http://issuu.com/geeadcps/docs/livro3_alta/81>. Acesso em: 7 out. 2015.

A escolha dos diagramas para representação da situação-problema, é que vai definir o modelo OO usado para o desenvolvimento do seu projeto.

6 EVOLUÇÃO DO PROJETO

Toda vez que se pensa em desenvolver um projeto de *software*, sempre há complicações quanto à introdução de novos campos em determinadas áreas do projeto. No projeto orientado a objeto, essa dificuldade já não é tão dificultada assim, e quando há uma inserção de um novo campo ou método no projeto, há uma grande facilidade de se lidar com isso sem que esta inserção afete parte do programa. Suponhamos que estamos desenvolvendo um projeto orientado a objetos, e o objeto do tipo Pessoa que estamos definindo precisa inserir o campo data Nascimento, sem muita complicação automaticamente no objeto do tipo Funcionário será incluído o campo data Nascimento, devido à interligação que um objeto tem com outro. Percebe-se pelo exemplo a robustez deste projeto, pois o projeto não foi afetado em nenhuma parte devido à inserção desse campo. (FONTE: Disponível em: <http://es1uenp.blogspot.com/2011/10/projeto-orientado-objetos.html>. Acesso em: 9 dez. 2015).

7 MODELAGEM ORIENTADA A OBJETOS

A modelagem orientada a objetos é uma forma de pensar sobre problemas reais usando a UML para representá-los.

Objetivos da modelagem

- Permitir uma visualização do sistema mais próxima da realidade.
- Permitir especificar arquitetura e comportamento de todas as funcionalidades do sistema.
- Fornecer padrões para desenvolver o sistema de forma organizada.
- Documentar as decisões tomadas durante o processo de desenvolvimento.

Por que modelar?

- Para entender mais facilmente as situações mais complexas.
- Modelos auxiliam o entendimento acerca do que será construído.

Muitas empresas de desenvolvimento iniciam a fase de codificação sem definir o modelo mais adequado, ou seja, sem ter um entendimento claro do que deve ser realmente feito; de qual é a real necessidade. O resultado é estouro do cronograma, alterações de escopo no meio do projeto e redundância de código. Nestas situações há a ausência de um plano de arquitetura de *software*, colocando o desenvolvimento do projeto como um todo, em risco.

A modelagem Orientada a Objetos é um processo organizado de produção de *software*, que utiliza técnicas predefinidas e notações convencionais. As etapas que compõem este processo correspondem ao ciclo de vida do *software*. Tradicionalmente, a formulação inicial do problema, a análise, o projeto, a implementação, os testes e a operação (manutenção e aperfeiçoamento) compõem estas etapas do ciclo de vida. "Um modelo é uma abstração de alguma coisa, cujo propósito é permitir que se conheça essa coisa antes de se construí-la" (RUMBAUGH, 1994).

Quatro princípios da modelagem visual:

- O modelo utilizado influencia na forma como o problema é abordado.
- Todo modelo se apresenta em vários níveis de precisão.
- Os melhores modelos são os que melhor refletem a situação na prática.
- Nenhum modelo único é suficiente.

RESUMO DO TÓPICO 1

- Projeto é algo temporário, com sequência de atividades com início, meio e fim. Seu resultado final fornece um produto ou serviço único e progressivo, tangível ou intangível limitado a restrições de tempo e custo. Cada projeto é único.

- Podemos considerar três etapas macro, no desenvolvimento de um projeto Orientado a Objetos:

  - A Análise Orientada a Objeto;
  - O Projeto Orientado a Objeto;
  - A Programação Orientada a Objetos.

- Projeto orientado a objeto é parte do desenvolvimento orientado a objeto.

- Representar fielmente as situações do mundo real, é a proposta da Orientação a Objetos. Por este motivo, o conceito não vê um sistema computacional como um conjunto de processos, mas como uma coleção de objetos que interagem entre si.

- Para viabilizar um **Projeto de *Software* Orientado a Objeto** é primordialmente necessário definir como vai ocorrer a comunicação entre o usuário e o sistema, e as principais funcionalidades que o sistema disponibilizará para o mesmo.

- A UML oferece diversos subsídios para a criação de modelos claros que nos auxiliem na construção de soluções de *software* de qualidade.

AUTOATIVIDADE

1 O que é um objeto?

2 Quais os benefícios do uso de modelos em projetos OO?

3 Quais os quatro princípios da modelagem visual?

TÓPICO 2

VISÃO GERAL DA CONSTRUÇÃO DE UM PROJETO ORIENTADO A OBJETOS; FASES E RESPECTIVAS ATIVIDADES

1 INTRODUÇÃO

O desenvolvimento de um projeto exige a adoção de uma metodologia que o organize em fases e atividades. Assim, o gerente de projetos e demais envolvidos conseguem gerenciar melhor as etapas, estimando de forma mais adequada, os tempos e alocando as pessoas corretas na customização das construções.

2 UMA VISÃO GERAL DA CONSTRUÇÃO DE UM PROJETO OO

Caro(a) acadêmico(a)! A seguir será exposto um roteiro genérico para que você tenha uma ideia de como se executa um projeto orientado a objetos. Para isso foi proposta uma metodologia de desenvolvimento, simulando uma situação real de desenvolvimento, que descreve os procedimentos relacionados ao produto final; como este, que será apresentado ao cliente e se atende aos padrões de qualidade.

2.1 A METODOLOGIA DE DESENVOLVIMENTO

Antes de iniciar qualquer atividade, você deverá montar um Plano de Execução do Projeto, para documentar e formalizar informações importantes do projeto.

2.1.1 Plano de Execução do Projeto

Objetivo: Descrever o perfil do cliente e identificar o serviço solicitado pelo mesmo, a fim de considerar os aspectos relacionados à gestão do projeto, bem como seu escopo, prazos e objetivos gerais.

Atividades:

1. Efetuar o primeiro contato com o cliente.
2. Entender a real necessidade do cliente.
3. Fazer o levantamento dos custos e recursos necessários para desenvolver o projeto.
4. Montar o cronograma inicial com as atividades a serem desenvolvidas e seus responsáveis, bem como o tempo disponível para execução e conclusão das mesmas.

Na maioria dos casos, estas atividades ficam sob responsabilidade do gerente de projetos ou do analista de sistemas.

Caso necessário, revise o Caderno de Estudos de Gestão de Projetos.

2.1.2 Levantamento de Requisitos

Finalizada a etapa do Planejamento do Projeto, inicia-se a fase de levantamento de requisitos do sistema para definir o seu funcionamento. Esta é considerada uma fase importantíssima, pois ela será o alicerce do escopo do projeto, norteando cronograma, custo e alocação de pessoas para o desenvolvimento.

Objetivos: fazer o mapeamento dos requisitos funcionais e não funcionais do sistema.

Para evitar redundância de informações no caderno, relembre os conceitos de requisitos no caderno de Engenharia de *Software*.

Atividades:

• Definição do Sistema: consiste em entender o que o cliente precisa em termos de solução de *software*. Deve-se procurar assimilar a situação na prática, para ter uma melhor visualização do fluxo das informações das quais ele precisa, facilitando assim os processos de análise, projeto e desenvolvimento do aplicativo. Deve-se documentar as funcionalidades que o sistema terá, bem como as expectativas do cliente em relação ao processo. Aqui também são mapeados os *Stakeholders*: todos os envolvidos no projeto.

• Identificação dos requisitos: o levantamento dos requisitos serve para definir o que deverá ser feito e os resultados esperados de cada atividade desenvolvida.

• Análise e classificação dos requisitos: consiste em separar os requisitos em dois grupos: funcionais e não-funcionais, identificando as inconsistências, riscos e viabilidade de implementação.

Sempre que necessário os requisitos deverão ser revistos ou esclarecidos junto ao cliente. Importante destacar que os requisitos devem ser formalizados. Para isso, deve ser criado um documento específico com esta finalidade. Este documento deve conter a assinatura do cliente e fará parte da documentação do projeto.

2.1.3 Mapeamento dos Casos de Uso

Objetivo: Construir os diagramas de Casos de Uso das funcionalidades do sistema

Atividades:

• **Construção do diagrama**: com base no levantamento de requisitos deverão ser construídos os Diagramas de Casos de Uso, que deverão contemplar os requisitos, interfaces e atores envolvidos.

• **Descrição em nível detalhado**. Deve-se efetuar o detalhamento dos requisitos. O analista de Sistemas é o responsável por esta etapa que utilizará uma ferramenta case para desenhar e documentar os diagramas.

Deve-se classificar os casos de uso pelo grau de prioridade dos mesmos. Isso é definido levando-se em conta o grau de importância/complexidade do caso de uso. A priorização dos casos de uso determina o número de iterações/ciclos de desenvolvimento para o sistema.

Havendo dúvidas em relação aos casos de uso, consulte o caderno de Análise Orientada a Objetos II.

2.1.4 Modelo Conceitual de Classes

Objetivo: Elaborar o Modelo Conceitual de Classes

Atividades: O modelo conceitual de classes deverá ser construído considerando-se:

- Conceitos (ou entidades) representados por classes.
- Associações, ou relacionamentos entre os conceitos.
- Multiplicidades.
- Atributos.
- Interfaces com sistemas já existentes.

Nesta fase, o analista de sistemas pode optar por criar um glossário do Modelo, padronizando termos para facilitar a comunicação e reduzir os riscos do projeto.

2.1.5 Diagrama de Estados/Atividades

Objetivo: Elaborar os Diagramas de Estados/Atividades do Sistema.

Atividades:

- **Diagrama de Estados:** Mostra os eventos e os estados do objeto e seu ciclo de vida. Usado em situações mais complexas, onde se faz necessário um refinamento do modelo, para maior compreensão da situação problema.
- **Diagrama de Atividades:** Mostra o fluxo de atividades do sistema de forma dinâmica. Tem como foco o fluxo de controle dos objetos.

Havendo dúvidas em relação ao Diagrama de Estados e Diagrama de Atividades, consulte o caderno de Análise Orientada a Objetos II.

2.1.6 Diagrama de Interação – (Colaboração e Sequência)

Objetivo: Elaborar os Diagramas de Interação (Colaboração e Sequência).

Atividades: Elaborar os diagramas de colaboração e sequência das atividades.

Não pode faltar nesta atividade:

- Diagrama de Atividades;
- Classes e instâncias;
- Ligações;
- Mensagens;
- Parâmetros e valores de retorno.

2.1.7 Diagrama de Classes do Projeto

Objetivo: Elaborar o Diagrama de Classes do Projeto.

Atividades: Diagrama de Classes

Para iniciar a construção do diagrama de Classes, o diagrama de Interação deverá estar concluído. Este diagrama exibe o detalhamento para as classes do projeto.

Um diagrama de classes de projeto deve conter:

- Classes, associações e atributos;
- Interfaces;
- Métodos;
- Informação de tipo de atributo;
- Navegabilidade;
- Dependências.

2.1.8 Esquema de Banco de Dados

Objetivo: Elaborar o Esquema do Banco de Dados.

Atividades: Montar a base de dados

A base de dados deverá conter:

- As entidades;
- Os atributos;
- Os domínios;
- As validações;
- Os relacionamentos;
- As *Views*;
- As *Stored procedures*;
- As *Triggers;*
- Funções.

Este artefato poderá ser construído na ferramenta DBDesigner.

Uma vez construído é possível elaborar um Dicionário de Dados, utilizando-se do recurso disponível na ferramenta.

2.1.9 Modelo Arquitetural

Objetivo: Elaborar o Modelo de Arquitetura do Sistema.

Atividades:

- Definir as formas de comunicação do projeto;
- Escolher as melhores tecnologias;
- Escolher o SGBD;
- Definir a arquitetura mais apropriada para cada projeto.

2.1.10 Construção Implementação

Objetivo: Construir o código do sistema.

Atividades:

- Fazer a codificação do sistema;
- Produzir a documentação desta etapa: especificações técnicas e funcionais, que serão utilizadas pelos programadores na geração adequada do código de programação.

2.2 SEGURANÇA

Objetivo: Estabelecer a política de segurança do sistema e suas informações.

Atividades:

- Implementar regras para o controle de acesso ao sistema;
- Tratar as situações de restrição de acesso a dados confidenciais;
- Determinar as permissões de acesso para as funcionalidades do sistema;
- Criar logs para registro de alteração ou exclusão de informações (log);
- Tratar a tolerância às falhas;
- Estipular as políticas de *backups*;
- Criar auditorias para evitar a violação das informações.

2.3 CONSTRUÇÃO TESTES

Objetivo: Elaborar Modelos de Testes.

Atividades:

Sempre que possível, elaborar o caderno de testes que servirá como roteiro para

- Identificar os objetos de teste e classificá-los;
- Reconhecer requisitos para cada tipo de teste;
- Definir o universo de teste;
- Fazer uma revisão ortográfica e gramatical do projeto todo;
- Corrigir defeitos e erros encontrados.

2.4 IMPLANTAÇÃO PLANO DE IMPLANTAÇÃO

Objetivo: Definição da fase de implantação.

Atividades:

- Definir a data da implantação.
- Definir os responsáveis pela implantação.
- Planejar as revisões e cargas das bases de dados.
- Revisar a documentação do projeto, principalmente os riscos.
- Revisar o modelo e seus requisitos.
- Checar o ambiente do cliente: estrutura, tecnologia.
- Certificar-se de que todos os responsáveis estarão presentes e acompanhando esta etapa.

2.5 PACOTE DE ENTREGA AO CLIENTE

Objetivo: Documentar e formalizar a entrega do projeto ao cliente.

Atividades:

Nesta etapa, estão previstas as atividades de:

- Elaboração de manuais do sistema;
- Elaboração da especificação de procedimentos de instalação do sistema;
- Gravação do sistema em mídia magnética/óptica;
- Criação do documento de termo de aceite do projeto que deverá ser assinado pelo cliente no momento da implantação do projeto.

2.6 TREINAMENTO

Objetivo: Elaborar plano e ministrar treinamento aos usuários do sistema.

Atividades:

Para esta fase destacam-se as seguintes atividades:

- Elaborar plano de treinamento;
- Determinar os participantes do treinamento e, caso necessário, dividi-los em equipes;
- Elaborar material: apostilas, cds e outros recursos necessários para o evento;
- Reservar local.

2.7 AVALIAÇÃO DO CLIENTE

Objetivo: Avaliar e mediar a qualidade do projeto.

Atividades:

Basicamente, consiste em três atividades principais:

- Elaborar plano de garantia do sistema;
- Avaliar como o sistema se comporta no cliente em um período de adaptação inicial;
- Definir plano para a manutenção corretiva e adaptativa.

Resumo das Atividades

As fases de execução e respectivas atividades estão resumidas no quadro a seguir:

QUADRO 2 – RESUMO DAS ATIVIDADES DE UM PROJETO GENÉRICO

Fases	Atividades
Planejamento	
Plano de Execução do Projeto	1. Contato Inicial com o Cliente; 2. Levantamento de Recursos do Projeto; 3. Cronograma Inicial.
Levantamento de Requisitos	1. Definição do Sistema; 2. Identificação dos Requisitos; 3. Análise e Classificação dos Requisitos.
Casos de Uso	1. Construção do Diagrama de Casos de Uso; 2. Descrição em Alto Nível; 3. Descrição em Nível Detalhado; 4. Priorização e Escalonamento dos Casos de Uso.
Construção	
Análise	1. Modelo Conceitual de Classes; 2. Glossário; 3. Diagramas de Estados/Atividades.
Projeto	1. Diagramas de Interação; 2. Diagrama de Classes do Projeto; 3. Esquema do Banco de Dados; 4. Modelo de Arquitetura.
Implementação	1. Implementação; 2. Segurança.
Testes	1. Testes.
Implantação	1. Plano de Implantação; 2. Pacote de Entrega ao Cliente; 3. Treinamento.
Avaliação do Cliente/Manutenção	1. Garantia da Qualidade.

FONTE: O autor

RESUMO DO TÓPICO 2

Neste tópico você acompanhou as etapas básicas de um projeto de *software* orientado a objetos.

AUTOATIVIDADE

Determine as fases do projeto e suas respectivas atividades.

UNIDADE 1

TÓPICO 3

PROJETO DE *SOFTWARE* ORIENTADO A OBJETOS; DECISÕES EM PROJETOS ORIENTADOS A OBJETOS

1 INTRODUÇÃO

O projeto de desenvolvimento de *software* é algo complexo por natureza e tem muitas características semelhantes aos projetos das áreas de engenharias quando consideramos o levantamento dos requisitos funcionais. No desenvolvimento de sistemas significa implementar os requisitos funcionais sob a forma de linguagens de programação (SILVA, 2007). Em projetos de *software* os requisitos traduzem as reais necessidades dos clientes. Podem ser entendidos como uma necessidade do próprio *software* para aumentar sua vida útil, ou alterações diversas motivadas pela própria evolução tecnológica.

Estes requisitos representam as entradas do sistema, ou seja, tudo o que é necessário para colocar em prática o seu funcionamento, de acordo com expectativas e necessidades pré-definidas. Porém, não são capazes de sozinhos, garantir todo o funcionamento. Entram em cena então os requisitos não funcionais, responsáveis por características como desempenho, usabilidade, suportabilidade, portabilidade, entre outras (MELO, 2006). Requisitos funcionais e não-funcionais conseguem traduzir necessidades e auxiliar toda a codificação do sistema no processo de desenvolvimento do *software*, o qual está inserido no ciclo de vida de desenvolvimento do aplicativo (LARMAN, 2007).

São vários os modelos de ciclo de vida de sistemas de *software*. Modelo cascata (ROYCE, 1970), espiral (BOEHM, 1989) e prototipação (GOMAA; SCOTT, 1981). O projeto é uma das atividades de todos os modelos de desenvolvimento conhecidos. O padrão do *Institute of Electrical and Electronics Engineers* (IEEE), IEEE 1074.1-1997 para a criação de modelos de ciclo de vida de *software* define que os principais processos da fase de desenvolvimento de *software* são: Requisitos, Projeto e Implementação (IEEE, 1997). Nestes modelos a atividade de projeto recebe como entrada a análise de requisitos que será transformada em regras e posteriormente documentada. Este documento será a base de analistas e programadores para iniciarem a execução do projeto de *software* com qualidade (PRESSMAN, 2005).

Ainda de acordo com o autor, algumas diretrizes são importantes para se desenvolver *software* com qualidade:

1. O *software* deve ser projetado em módulos;

2. O sistema deve manter distintamente representações para dados, arquitetura, interfaces e componentes;

3. Um projeto deve prover componentes que possuam características de independência funcional;

4. Um projeto deve prover interfaces que reduzam a complexidade das conexões entre os componentes e o ambiente externo.

Dando sequência à preocupação com a qualidade, o autor cita outros aspectos importantes nos projetos de desenvolvimento de *software*, que devem ficar sob responsabilidade dos projetistas de *software*:

1. O projeto deve ser rastreável para o modelo de análise;

2. Sempre considere a arquitetura do sistema a ser construído;

3. O projeto dos dados é tão importante quanto o projeto das funções de processamento;

4. As interfaces (ambas internas e externas) devem ser projetadas com cuidado;

5. A interface com o usuário deve estar sintonizada com as necessidades do usuário final;

6. O projeto, no nível dos componentes, deve ser funcionalmente independente;

7. Componentes devem ser fracamente acoplados entre eles e com o ambiente externo;

8. As representações (modelos) de projeto devem ser entendidas facilmente;

9. O projeto deve ser desenvolvido iterativamente.

Estar ciente da complexidade e do tamanho do projeto a ser desenvolvido é importante para garantir a qualidade do mesmo. Determinar o tamanho do que deve ser feito é crucial para que se consiga estimar os recursos necessários para a customização das atividades. Para auxiliar neste levantamento existem métricas de *software*, que são distintas maneiras de se medir o tamanho de um projeto de *software*. Uma das mais utilizadas é a contagem de linhas de código (LOC) (PRESSMAN, 2005), devido à facilidade em ser computada.

Existem outras formas de medida, como:

- Complexidade de programa (PETERS; PEDRYCZ, 2001);
- Complexidade ciclomática (McCABE, 1976);
- Complexidade da arquitetura como DSQI (*Design Structure Quality Index*);
- Complexidade orientada a objetos como as propostas por Lorenz e Kidd (LORENZ; KIDD, 1994) e por Chidamber e Kemerer (CHIDAMBER; KEMERER, 1994);
- Complexidade das funcionalidades do *software* como pontos por função (VAZQUEZ; SIMÕES; ALBERT, 2003);

- Pontos por casos de uso (ANDA et al., 2001), entre outras.

Todas as métricas, sem exceção, avaliam o tamanho e a complexidade do projeto, servindo de alicerce para elaboração dos cronogramas no que se refere aos tempos de desenvolvimento e pessoas alocadas em cada tarefa, além de permitir o acompanhamento da produtividade das equipes.

Estas métricas serão abordadas no Tópico 1 da Unidade 2, com foco para as métricas voltadas para o desenvolvimento Orientado a Objetos.

2 DESENVOLVIMENTO ORIENTADO A OBJETOS E O PROCESSO UNIFICADO

Conforme visto no Tópico 1, desenvolvimento orientado a objetos começou em 1967 com a linguagem Simula-67 (BIRTWISTLE et al., 1975) seguida pela linguagem Smalltalk (GOLDBERG; ROBSON, 1989) e C++ (STROUSTRUP, 1997) entre outras. Nos anos 80 começaram a surgir as metodologias de desenvolvimento orientadas a objetos. Num período de 6 anos (entre 1989 e 1994) surgiram aproximadamente 50 métodos de desenvolvimento orientados a objetos. Esta fase ficou conhecida como a guerra dos métodos (BOOCH; RUMBAUGH; JACOBSON, 2005).

O resultado disso foi a criação da UML, apresentada, na sua versão 1.0, em 1997 (BOOCH; RUMBAUGH; JACOBSON, 2005). É uma metodologia muito útil para visualizar, especificar, construir e documentar os requisitos necessários aos projetos de *software*, sendo considerada desde 1997 como padrão internacional pelo OMG (*Object Management Group*) (OBJECT MANAGEMENT GROUP, 2005).

Outra consequência foi a unificação das metodologias pela Rational, que usa a UML em seus modelos e o define como Processo Unificado (JACOBSON; BOOCH; RUMBAUGH, 1999).

O Processo Unificado, apesar de não ser um padrão, é amplamente adotado, sendo considerado como um modelo de processo de desenvolvimento de *software* orientado a objetos.

2.1 PROCESSO UNIFICADO

O processo unificado tem como principal característica o ciclo de vida iterativo, onde cada fase do desenvolvimento é dividida em iterações (KRUCHTEN, 2003). Uma iteração também pode ser entendida como fase, sendo que cada uma é construída, testada, validada e integrada aos demais módulos do projeto. Esta forma de trabalho auxilia na construção de sistemas complexos, pois é mais flexível para assimilar novos requisitos ou mudanças de escopo nos requisitos existentes, além de permitir a identificação e correção de falhas ou erros mais cedo no processo de desenvolvimento (KRUCHTEN, 2003).

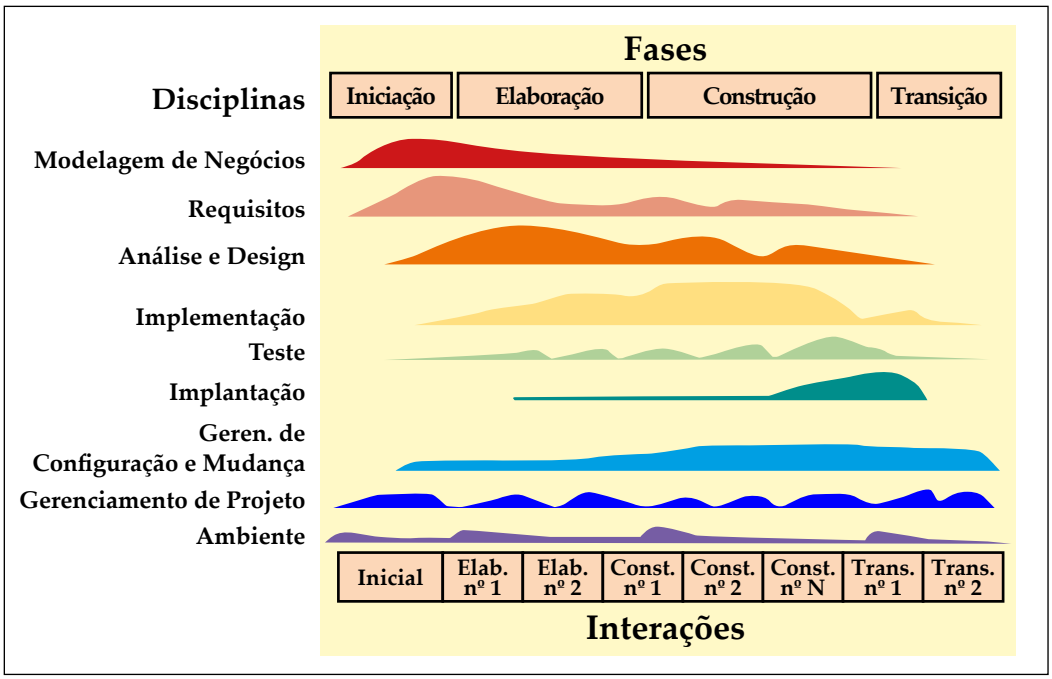
O processo unificado tem seu foco nos casos de uso e toda a construção do sistema é feita com base neles, organizando os requisitos funcionais em função do resultado esperado pelo usuário. O processo unificado também é centrado na arquitetura cujo desenho é traçado no início do projeto e evolui a cada fase de desenvolvimento. Isso dificulta distribuir o projeto para várias equipes, mas em consequência, há redução do retrabalho, facilitação do reuso dos componentes e agilidade no gerenciamento do projeto (JACOBSON; BOOCH; RUMBAUGH, 1999).

2.2 FASES DO PROCESSO UNIFICADO

Fases e Fluxos são as dimensões do processo unificado (KRUCHTEN, 2003), conforme ilustrado na figura a seguir. A dimensão das fases exibe características do ciclo de vida do Processo Unificado em função do tempo. Uma fase é considerada um marco do projeto onde determinados objetivos são alcançados e decisões são tomadas em relação à fase seguinte. (BOOCH; RUMBAUGH; JACOBSON, 2005).

Cada fase, porém, tem seu ciclo de desenvolvimento, suas atividades específicas, requisitos e decisões. Este ciclo também envolve desde a análise dos requisitos até os testes finais antes da mesma ser considerada finalizada para ser acoplada ao restante do projeto (KRUCHTEN, 2003). As fases do processo unificado são: concepção (iniciação), elaboração, construção e transição, conforme se pode ver na figura a seguir.

FIGURA 5 – FASES DO PROCESSO UNIFICADO



FONTE: Kruchten (2013)

A fase da concepção é de extrema importância, pois é ela que define o escopo, cronograma e distribuição das atividades. Este planejamento será a referência do projeto até que o mesmo seja concluído. É uma etapa que envolve muito o usuário final. Aqui se faz a documentação dos requisitos, que deverão ser confirmados pelo usuário. Nesta fase elabora-se também o esboço inicial dos casos de uso do sistema.

A fase da elaboração faz o refinamento dos casos de uso desenhados na fase de concepção, além de preparar o ambiente para a fase de desenvolvimento do aplicativo. Esta fase geralmente é encarregada da finalização dos casos de uso, a descrição dos requisitos não funcionais, definição da arquitetura de *software* e prototipação do aplicativo (KRUCHTEN, 2003).

A fase de construção limita-se a elaborar a codificação do sistema e a fase de transição prepara a entrega do produto ao cliente, bem como os manuais. Também define a fase de treinamento, acompanhamento pós-entrega, ajustes finais e correções de erros (JACOBSON; BOOCH; RUMBAUGH, 1999).

3 PROBLEMAS RELACIONADOS AO PROJETO DE *SOFTWARE* – DECISÕES DO PROJETO

Várias são as dificuldades com as quais as equipes de desenvolvimento de *software* se deparam: não cumprimento dos requisitos, falhas nos cronogramas, dificuldades ao elaborar o projeto, rotatividade de profissionais, rápida evolução tecnológica, entre outros. Isso acontece por falhas cometidas durante o processo de desenvolvimento, como: falhas no levantamento dos requisitos funcionais, inconsistências de projeto não encontradas antecipadamente, poucos testes realizados, má administração dos riscos, estimativas inadequadas de custos e prazos, entre outras (SILVA, 2007).

Outros empecilhos:

- Definição e planejamento inadequado do projeto;
- Práticas de gerenciamento inadequado de mudanças de escopo;
- Não gerenciamento do plano de trabalho (*Workplan*);
- Falta de comunicação com os interessados;
- Falta de um bom gerenciamento de qualidade.

Com cenários turbulentos, gestores e equipes necessitam tomar decisões rapidamente para mudar a rota do projeto, sempre que riscos mais graves se apresentam no processo de desenvolvimento. Decidir faz parte do processo de customização dos aplicativos.

Decidir é algo importante, complexo e não sistemático, uma vez que estas decisões geralmente são baseadas em conhecimentos prévios de analistas, projetistas e programadores, sem o uso de modelos específicos para as situações que se apresentam no cotidiano do desenvolvimento. O desenvolvimento de

software é um processo onde cada pessoa envolvida constantemente toma decisões, tanto técnicas quanto gerenciais. Por isso, o entendimento total do projeto é fator imperativo para uma tomada de decisão acertada.

A maioria dos gestores de projetos de *software* toma decisões reativas, ou seja, como reflexo para algo que aconteceu no momento, sem se preocupar com a relação da situação com o projeto de forma geral. Isso gera riscos para o futuro. São decisões imediatistas que podem gerar prejuízos ao projeto, causando falhas críticas ao mesmo.

Geralmente, decisões em projetos de *software* são tomadas rapidamente, pois ocorrem em cronogramas apertados mal definidos em decorrência de um precário levantamento de requisitos. Isto normalmente ocorre em uma organização com pouca maturidade de capacidade do processo de *software*, algumas decisões realmente devem ser tomadas rapidamente a fim de não atrapalhar o progresso de desenvolvimento, como realocação de pessoas em atividades, bem como decisões que envolvam a priorização das mesmas. As decisões que envolvem os riscos do projeto são as mais críticas, principalmente quando relacionadas a riscos que não foram previstos (BOOCH, 2006). Essas decisões podem gerar arquiteturas.

Estas arquiteturas se manifestam de duas formas: são intencionais ou acidentais. A arquitetura intencional surge por uma necessidade e seu desenvolvimento ocorre de forma explícita – existe intenção – na sua construção. Já a arquitetura acidental ganha forma com base em decisões que ocorrem durante o ciclo de desenvolvimento dos projetos (BOOCH, 2006). Conforme for sendo validada, uma arquitetura acidental pode vir a se tornar uma arquitetura intencional no projeto em desenvolvimento ou em projetos futuros.

Para Pressmann (2005), projetar *software* é uma atividade complexa. Grande parte desta complexidade é inerente à própria natureza do *software*. Sistemas de *software* são mais complexos proporcionalmente ao seu tamanho, que qualquer outra construção humana. Computadores são algumas das máquinas mais complexas já construídas pelo homem. Eles possuem um número muito grande de estados. Sistemas de *software* possuem algumas ordens de grandeza a mais de estados que computadores (BROOKS, 1987). Além de serem complexos por natureza, sistemas de *software* são constantemente sujeitos a pressões por mudança. A maior pressão por mudanças vem da necessidade de modificação da funcionalidade do sistema que é incorporada pelo *software*.

Outra razão para essa pressão por mudanças se deve ao fato do *software* ser uma entidade flexível e que pode ser mudada com muito mais facilidade do que produtos manufaturados (BROOKS, 1987). Para tentar administrar essa complexidade inerente de sistemas de *software*, emergiu um conceito chamado Independência Funcional (STEVENS; MEYERS; CONSTANTINE, 1974). Este conceito está intimamente ligado a

modularidade, ocultação de informações e abstração (PRESSMAN, 1995). Em sistemas de *software*, a Independência Funcional pode ser medida através de dois critérios: coesão e acoplamento. "Coesão é uma medida da força funcional relativa de um módulo" (PRESSMAN, 1995). Em outras palavras a coesão mede o grau com que as tarefas executadas por um único módulo se relacionam entre si (IEEE, 1990).

Para *software* orientado a objetos, coesão também pode ser conceituada como sendo o quanto uma classe encapsula atributos e operações que estão fortemente relacionados uns com os outros (PRESSMAN, 2005). "Acoplamento é uma medida da interdependência relativa entre os módulos" (PRESSMAN, 1995). Para sistemas de *software* orientados a objetos pode-se definir acoplamento como sendo o grau com o qual classes estão conectadas entre si (PRESSMAN, 2005). Existem métricas definidas na literatura para coesão de um módulo (BIEMAN; OTT, 1994) e acoplamento entre módulos (DHAMA, 1995).

As métricas de coesão e acoplamento tratam de avaliar como os componentes dependem uns dos outros. Quando se escolhe um conjunto de requisitos funcionais muito interdependente, os componentes da solução gerada tendem a ser interdependentes também. O desenvolvimento de *software* orientado a objetos tem como um dos seus preceitos aumentar a coesão e diminuir o acoplamento entre os módulos do sistema.

O desenvolvimento orientado a objetos facilita para o desenvolvedor criar componentes mais reutilizáveis (COLEMAN et al., 1996). Para isso, a orientação a objetos disponibiliza abstrações como classes, objetos, interfaces, atributos e métodos (WINCK; GOETTEN, 2006). Além disso, a orientação a objetos introduziu conceitos como encapsulamento, herança e polimorfismo para aumentar a reutilização e a extensibilidade e facilitar a manutenção de sistemas de *software* (COLEMAN et al., 1996). Entre as decisões mais importantes em um projeto orientado a objetos estão a identificação das classes que comporão o sistema e a atribuição de responsabilidades para essas classes. Para tentar ajudar o projetista nessa tomada de decisões foram criados padrões. Um padrão é um par composto de um problema e a respectiva solução, que possui um nome e que pode ser aplicado em novos contextos, contendo diretrizes de como aplicá-lo a essas novas situações (LARMAN, 2004). O padrão tem por objetivo permitir que o desenvolvedor reutilize soluções já aplicadas e aceitas como boas soluções. Esta reutilização de soluções tem por objetivo diminuir o tempo e o esforço de desenvolvimento (PRESSMAN, 2005). Segundo Gamma et al. (2000), "Padrões de projeto tornam mais fácil reutilizar arquiteturas e experiências bem-sucedidas" (GAMMA et al., 2000).

Essas soluções podem ser tanto arquiteturas intencionais como acidentais. No caso de arquiteturas acidentais, essas soluções com o tempo

se transformam em arquiteturas intencionais (BOOCH, 2006), ou então, são aceitas até que deixem de ser soluções interessantes. Os padrões de projeto representam soluções de projeto que se mostraram eficientes. No caso de soluções acidentais, muitas vezes elas surgem de decisões de projeto tomadas sem a aplicação de critérios mais precisos. Em certos casos, um padrão de projeto pode representar uma arquitetura acidental, que não necessariamente representa uma solução ótima. Por outro lado, a orientação a objetos fornece um modo eficaz de representar os elementos inerentes ao domínio do negócio como dos componentes da solução (WINCK; GOETTEN, 2006). As abstrações como classes e objetos permitem que o desenvolvedor crie componentes da solução que representem entidades do mundo real relativas ao negócio a ser automatizado.

No entanto, não disponibiliza um modo fácil quando se quer representar elementos que não estão diretamente relacionados com o negócio no mundo real, mas fazem parte da solução (WINCK; GOETTEN, 2006). Esses elementos servem muitas vezes como suporte à solução, como por exemplo o tratamento de exceções. As características relevantes de uma aplicação, agrupadas por similaridade são chamadas de interesses (*concerns*) (WINCK; GOETTEN, 2006). Os interesses representam requisitos do sistema. Um tipo de interesse fica totalmente contido em um ou dois componentes do sistema. Esse tipo de interesse normalmente está fortemente relacionado com o domínio da aplicação.

Tanto o código emaranhado quanto o espalhamento de código prejudicam a coesão e o acoplamento do sistema. O código emaranhado diminui a coesão porque quando uma classe implementa vários interesses, o grau com que as tarefas realizadas por esta classe estão relacionadas diminui e esta classe poderia ser dividida em classes mais coesas. Por outro lado, o código espalhado aumenta o acoplamento do sistema pois qualquer mudança na implementação do interesse implica na alteração de várias classes. O aparecimento de código espalhado e código emaranhado na implementação de um sistema ocasiona os seguintes problemas: replicação do código, dificuldade de manutenção, redução da capacidade de reutilização de código e aumento da dificuldade de compreensão do código (WINCK; GOETTEN, 2006). Outro tipo de interesse transversal é a extensão.

Uma extensão representa um serviço ou característica adicional a ser implementada no sistema (JACOBSON; NG, 2004). O conceito de interesse de extensão é análogo ao conceito de extensão de casos de uso, onde "o caso de uso que estende define um conjunto modular de incrementos de comportamento que aumentam uma execução do caso de uso estendido sob condições específicas" (OBJECT MANAGEMENT GROUP, 2005). Em outras palavras, um interesse de extensão aumenta o comportamento de outro interesse sob determinadas condições. Isto gera a necessidade de se colocar um fragmento de código onde não era necessário antes. É o chamado

código cola (*glue code*) (JACOBSON; NG, 2004). Adicionar este tipo de código torna as classes originais difíceis de serem compreendidas e as extensões difíceis de serem identificadas à primeira vista (JACOBSON; NG, 2004). A inabilidade do desenvolvimento orientado a objetos em representar de forma modular interesses transversais, pode estar relacionada com uma falta de critérios mais precisos na hora de identificar e selecionar o conjunto de requisitos funcionais adequado.

No desenvolvimento orientado a objetos usando a UML (OBJECT MANAGEMENT GROUP, 2005) e o processo unificado (JACOBSON; BOOCH; RUMBAUGH, 1999), os requisitos funcionais são capturados e organizados através dos casos de uso. Os casos de uso foram introduzidos por Jacobson (JACOBSON, 1987) e se tornaram uma técnica popular entre os desenvolvedores. Um caso de uso é uma descrição de um conjunto de ações, incluindo variantes, que um sistema executa para produzir um resultado de valor observável para o ator (BOOCH; RUMBAUGH; JACOBSON, 2005). A modelagem de casos de uso é uma técnica que captura e organiza os requisitos funcionais em função do resultado produzido para o usuário. Devido a este fato, usar os casos de uso para capturar e organizar os requisitos funcionais pode gerar muitas dificuldades para lidar com interesses transversais já que estes representam elementos que não estão diretamente relacionados com o negócio no mundo real e servem muitas vezes como suporte à solução. Para tentar amenizar essa dificuldade (JACOBSON; NG, 2004) propuseram novos tipos de casos de uso como os casos de uso de utilidade (*utility use cases*) e os casos de uso de infraestrutura (*infraestructure use cases*).

Casos de uso de utilidade representam funcionalidades que não estão diretamente relacionadas com a interação com o usuário mas servem como suporte à execução do sistema. Casos de uso de infraestrutura representam requisitos não funcionais do sistema que necessitam de tratamento pelo sistema. A escolha incorreta do conjunto de requisitos funcionais para o projeto, pode causar muitos problemas para o desenvolvimento do *software*. Ela pode prejudicar a independência funcional dos componentes que é uma diretriz de um bom projeto. Portanto, a falta de critérios para escolher um conjunto adequado de requisitos funcionais pode comprometer a qualidade do produto de *software* a ser gerado pelo projeto.

FONTE: Pimentel (2007, p. 38-46).

LEITURA COMPLEMENTAR

A IMPORTÂNCIA DAS MÉTRICAS PARA O CONTROLE DE PROJETOS DE *SOFTWARE*

Daniel Trindade Ituassú

Os avanços da tecnologia ocorridos nas últimas décadas têm proporcionado aos gestores de negócios um grande volume de ferramentas que os auxiliam na tomada de decisões. Soluções em Tecnologia da Informação têm surgido de forma acelerada fazendo com que o mercado de desenvolvimento de *software*, ao mesmo tempo em que se torna um dos que mais cresce no país, seja também um dos mais concorridos.

Esse novo cenário de negócios tem obrigado as empresas desenvolvedoras a buscarem constantemente tecnologias e métodos que lhes permitam garantir a qualidade dos produtos oferecidos para que não percam competitividade no mercado em que atuam. Neste contexto, a utilização de métricas que possam definir o desempenho dos projetos, bem como dos produtos resultantes deles, tem assumido um papel cada vez mais importante no setor e, consequentemente, no meio acadêmico, que se prepara para enfrentar os desafios propostos por estas organizações. No entanto, conforme poderá ser observado no presente trabalho, a definição de métodos de medição apenas não é suficiente para a garantia da qualidade.

Faz-se necessário também o devido monitoramento e controle de todos os processos e indicadores resultantes deles.

Palavras-chaves: Projetos, PMI, Infraestrutura, Escopo, Gestão, RUP, Métricas, Qualidade, Monitoramento e Controle.

1 INTRODUÇÃO

Os últimos 20 anos foram marcados por grandes avanços tecnológicos que proporcionaram ao mundo atual um cenário empresarial muito mais dinâmico e ávido por ferramentas que auxiliem os gestores nas tomadas de decisões. O resultado disto foi o aumento do número de empresas investindo em Tecnologia da Informação e, consequentemente, da quantidade de empresas desenvolvedoras de *software*. Isto porque, segundo Rezende (2002, p. 122), "a formulação de estratégia de qualquer negócio é feita a partir de informações disponíveis e, portanto, nenhuma estratégia consegue ser melhor do que a informação da qual é derivada".

Diante deste contexto, a indústria de *software* tem sido uma das que mais têm crescido nos últimos anos. Segundo Mizuno (1993), a demanda pelo *software* tem apresentado um crescimento de até cerca de 10 vezes por década gerando um

ambiente de alta competitividade entre as empresas do setor. Neste contexto, as empresas desenvolvedoras vêm se preocupando cada vez mais com a qualidade dos produtos oferecidos. Tal fato tem feito com que novas técnicas e ferramentas fossem sendo desenvolvidas com a finalidade de melhorar o desempenho das organizações que desenvolvem *software* (FENTON, 1994).

Dentre as tecnologias que vêm sendo desenvolvidas, encontram-se as de gerenciamento de projetos de desenvolvimento. Isto porque, segundo Boehm (1987), um gerenciamento ineficiente pode gerar custos desnecessários ao empreendimento. Estes, por sua vez, serão convertidos em preços mais altos fazendo com que a empresa desenvolvedora perca competitividade no mercado.

No entanto, para avaliar a qualidade de um produto, faz-se necessária a identificação e definição de meios de medi-la para que se quantifique o grau de alcance da qualidade. Diante disto, para computar uma característica de qualidade, é essencial que se estabeleçam métricas capazes de quantificá-la e fazer uma medicação para determinar a medida, resultado da aplicação métrica. (DUARTE, 2000).

2 REFERENCIAL TEÓRICO

[...]

2.2 MÉTRICAS DE DESENVOLVIMENTO DE *SOFTWARE*

Definidos os processos que serão utilizados no desenvolvimento de um software, pode-se, então, definir as métricas que deverão ser adotadas para o acompanhamento do projeto e para verificar se os critérios de qualidade estabelecidos estão sendo alcançados, bem como os custos e os prazos estipulados. Segundo Mello Filho (2002, p. 4): "*Atributo mensurável de uma entidade. Define-se métrica primitiva como sendo aquele atributo mensurável que pode, por leitura direta, ser obtido da própria entidade. Define-se métrica derivada como aquele atributo mensurável que é obtido por cálculo a partir de métricas primitivas". (MELLO FILHO, 2002, p. 4)*

Pode-se dizer, portanto, que as métricas servem para administração e controle do processo de desenvolvimento para determinar os resultados obtidos. Pressman (1987) cita algumas razões pelas quais é importante medir o *software*:

- Indicar a qualidade do produto.
- Avaliar a produtividade das pessoas que produzem o produto.
- Avaliar os benefícios (em termos de qualidade e produtividade) derivados de novos métodos e ferramentas de *software*.
- Formar uma linha básica para estimativas.
- Ajudar a justificar os requerimentos de novas ferramentas ou treinamento adicional.

Segundo Mello Filho (2002, p. 6):

> A visão do RUP sobre métricas parte de uma questão fundamental: por que medimos? A primeira e mais importante resposta é que medimos para ganhar controle sobre o projeto e, portanto, para sermos capazes de gerenciá-lo. Secundariamente medimos para melhor estimar novos projetos. Finalmente, medimos para saber como estamos evoluindo em determinadas áreas de nosso projeto ou Organização. (MELLO FILHO, 2002, p. 6).

Os benefícios contemplados pela implementação de métricas no desenvolvimento de *software* são a possibilidade de indicar a qualidade do produto, avaliar a produtividade das pessoas que trabalham na produção e formar uma linha básica de estimativas e também ajudar a justificar os pedidos de novas ferramentas e treinamentos para a equipe alocada no projeto. (PRESSMAN, 1995). Como exemplo de métrica que pode ser utilizada no desenvolvimento de *software,* pode ser citado o número de solicitações de mudanças, que indicará como está o andamento do gerenciamento do escopo do projeto e determinará a estabilidade dos requisitos de um projeto.

Segundo RUP (2000), podemos categorizar os objetivos da medição da seguinte forma:

• **Objetivos da organização:** Os objetivos de uma organização precisam, geralmente, estar relacionados aos seus custos por produto, tempo de desenvolvimento e qualidade dos seus produtos. Como exemplos de métricas relevantes à organização, podemos citar: custo por ponto de função, custo por caso de uso, custo por linha de código, esforço por linha de código ou por ponto de função, defeitos por linha de código.

• **Objetivos dos Projetos:** Projetos devem ser entregues com os requisitos funcionais e não-funcionais dos seus clientes atendidos, no prazo e orçamento definidos e atendendo às restrições do cliente. Para tal, é necessário o uso de métricas para responder a perguntas como: O projeto está atendendo aos *milestones* definidos? Como estão os esforços e custos atuais do projeto comparados ao planejado? Os requisitos funcionais estão completos? O sistema está atendendo ao desempenho desejado? Além de definir as métricas para auxiliar nessas questões, é preciso definir interpretações e ações a serem tomadas com base nos resultados das métricas, mas isso só poderá ser possível a partir de avaliações dos dados históricos de projetos passados.

• **Objetivos Técnicos:** Definem métricas para as necessidades técnicas, são de características estruturais e comportamentais. Alguns exemplos dessas métricas são: frequência de mudança de requisitos (volatilidade), requisitos corretos e completos (validade, completude), projeto realizando todos os requisitos (completude), estabilidade da arquitetura, extensão dos testes para cobrir todo o sistema, incidência de defeitos por atividade e fases. Mello Filho (2002) afirma ainda que muitas métricas podem ser desenvolvidas e utilizadas, mas em se tratando de desenvolvimento de *software*, existem sete métricas básicas para qualquer projeto deste tipo. Mello Filho (2002) afirma que estes indicadores

têm a virtude de serem simples, fáceis de coletar e de interpretar e são divididos em dois grupos, indicadores de gerenciamento e de qualidade.

Indicadores de Gerenciamento:

- Trabalho e Progresso.
- Orçamento e Despesas.
- Alocação e Rotatividade na Equipe.

Indicadores de Qualidade:

- Fluxo de Mudanças e Estabilidade.
- Fragmentação e Modularidade.
- Retrabalho e Adaptabilidade.
- Tempo médio entre falhas e Maturidade.

Vale lembrar, no entanto, que para se definir as métricas de qualidade, a empresa deve, inicialmente, identificar quais são os atributos do produto mais valorizados pelos clientes e como tornar os atributos do produto da empresa mais atrativos, em relação aos dos concorrentes. Isto porque nem sempre os critérios estabelecidos pela organização são os mesmos identificados como importantes pelo cliente.

2.3 MONITORAMENTO E CONTROLE

Tão importante quanto a definição de métricas é o monitoramento e controle das mesmas. Afinal, de nada adianta definir indicadores de desempenho para um projeto se estes não forem acompanhados para que seja feita uma análise do projeto e do produto desenvolvido. Esta fase do projeto tem como foco principal assegurar que as atividades, os prazos, os custos e os demais parâmetros estabelecidos durante o planejamento do projeto, sejam cumpridos e forneçam subsídios ao Gerente de Projeto para que ele possa tomar suas decisões. Vargas (2005, p. 34) afirma que: *"É a fase que acontece paralelamente ao planejamento operacional e à execução do projeto. Tem como objetivo acompanhar e controlar aquilo que está sendo realizado pelo projeto, de modo a propor ações corretivas e preventivas no menor espaço de tempo possível após a detecção da anormalidade"*.

Almeida (2006) ainda afirma que essa fase é caracterizada por:

- Controle da qualidade.
- Controle de alterações do escopo e qualidades.
- Atualizações e revisões dos cronogramas físicos e financeiros.
- Retroalimentação do planejamento.

Pode-se dizer, portanto, que na fase de monitoramento é onde serão acompanhadas as métricas estabelecidas na fase de planejamento e, portanto, sendo a etapa do projeto essencial para a garantia da sua qualidade.

FONTE: Disponível em: <http://www.techoje.com.br/site/techoje/categoria/detalhe_artigo/678>. Acesso em: 25 fev. 2016.

RESUMO DO TÓPICO 3

Neste tópico você aprendeu que:

- O projeto de desenvolvimento de software é algo complexo por natureza e tem muitas características semelhantes aos projetos das áreas de engenharias quando consideramos o levantamento dos requisitos funcionais.

- O processo unificado tem como principal característica o ciclo de vida iterativo, onde casa fase do desenvolvimento é dividida em iterações. Uma iteração também pode ser entendida como fase, sendo que cada uma é construída, testada, validada e integrada aos demais módulos do projeto.

- Esta forma de trabalho auxilia na construção de sistemas complexos, pois é mais flexível para assimilar novos requisitos ou mudanças de escopo nos requisitos existentes, além de permitir a identificação e correção de falhas ou erros mais cedo no processo de desenvolvimento.

- Várias são as dificuldades com as quais as equipes de desenvolvimento de software se deparam: não cumprimento dos requisitos, falhas nos cronogramas, dificuldades ao elaborar o projeto, rotatividade de profissionais, rápida evolução tecnológica, entre outros. Isso acontece por falhas cometidas durante o processo de desenvolvimento, como: falhas no levantamento dos requisitos funcionais, inconsistências de projeto não encontradas antecipadamente, poucos testes realizados, má administração dos riscos, estimativas inadequadas de custos e prazos.

AUTOATIVIDADE

Cite as principais dificuldades das equipes de desenvolvimento de projeto orientado a objetos.

UNIDADE 2

MÉTRICAS DE *SOFTWARE*

OBJETIVOS DE APRENDIZAGEM

Esta unidade tem por objetivos:

- relembrar conceitos e tipos de métricas de projetos de software tradicionais;
- aprofundar os conhecimentos em métricas de projetos de software orientado a objetos;
- conhecer os conceitos, características e tipos de padrões de projetos orientados a objetos.

PLANO DE ESTUDOS

Esta unidade de ensino contém três tópicos, sendo que no final de cada um, você encontrará atividades que contribuirão para a apropriação dos conteúdos.

TÓPICO 1 – MÉTRICAS DE *SOFTWARE*

TÓPICO 2 – MÉTRICAS PARA PROJETOS ORIENTADOS A OBJETOS

TÓPICO 3 – PADRÕES DE PROJETO: CARACTERÍSTICAS E TIPOS

TÓPICO 1

MÉTRICAS DE *SOFTWARE*

1 INTRODUÇÃO

Gerenciar produtividade e qualidade em desenvolvimento de *software* pode ser considerada uma tarefa árdua e tecnicamente impossível de ser realizada sem suporte tecnológico e metodológico. Como alicerce neste processo, surgem as métricas de *software*, que são capazes de aferir as atividades e também apontam tendências no processo de desenvolvimento. De acordo com Ambler (1998), a gestão de projetos de *software* somente é capaz de atingir maturidade e eficácia quando existem medidas que possibilitam o gerenciamento das finanças e da produtividade no desenvolvimento dos *softwares*.

As medidas ou métricas são métodos da Engenharia de *Software* capazes de determinar o tamanho dos projetos de maneira consistente e transformá-los em números. Além disso, as métricas permitem otimizar os esforços do projeto, bem como possibilitar a escolha ou substituição dos recursos necessários ao desenvolvimento do mesmo. Entenda por recursos: pessoas, metodologias, tecnologias etc.

Geralmente, a produtividade no desenvolvimento de projetos de *software* fica abaixo do esperado. Por isso, qualquer atividade que pode ser controlada, deve ser medida. Isso permite que os gestores do projeto (gerentes de projeto e analistas de sistemas), tenham informações confiáveis para poder planejar e redirecionar as ações com maior certeza nas decisões e menores riscos para o desenvolvimento das atividades.

As métricas de *software* permitem que se desenvolvam aplicativos mais complexos com maior velocidade, qualidade e menor custo. As técnicas de medições baseadas em objetos, simplificam e agilizam o projeto de atividades mais complexas. Este é um dos motivos que faz com que as organizações migrem suas aplicações para projetos Orientados a Objetos.

Ambler (1998) lista algumas métricas que podem ser utilizadas em *softwares* orientados a objetos e que serão detalhadas adiante:

a) contagem de métodos;
b) número de atributos de instância;
c) profundidade da árvore de herança;
d) número de filhos;

e) utilização global;
f) quantidade de atributos de classe;
g) tamanho do método;
h) resposta do método;
i) comentários por método.

2 MÉTRICAS

Existem basicamente duas categorias de métricas de *software*, de acordo com Pressman (1995):

• **Medidas diretas:** estão relacionadas ao custo e esforços aplicados no desenvolvimento das atividades.

• **Medidas indiretas:** remetem aos aspectos intangíveis, como: funcionalidade, qualidade, complexidade, eficiência, sendo estas mais difíceis de serem medidas.

Pressman (1995) ainda classifica as métricas da seguinte forma:

• **métricas orientadas ao tamanho**: usam como referência para medição as linhas de código, esforço, custo, quantidade de documentação;

• **métricas orientadas à função**: as medidas concentram-se na funcionalidade ou qualidade do *software*, um exemplo destas métricas é a técnica de Análise por Pontos de Função;

• **métricas orientadas às pessoas**: usam como medida a forma de trabalho das pessoas e em como elas desenvolvem os aplicativos.

2.1 A ORIGEM DOS SISTEMAS MÉTRICOS

Os sistemas métricos surgiram em meados de 1970 para medir indicadores do processo de desenvolvimento de sistemas.

Moller e Paulish (1993) destacam as tendências das métricas, quando as mesmas surgiram:

• **medida de complexidade do código:** eram fáceis de obter, desde que calculados pelo próprio *software* automatizado;

• **estimativa do custo de um projeto de *software***: focava no trabalho e o tempo gasto para se desenvolver um *software*. Tinha como base a quantidade de linhas de código necessárias para a implementação;

• **garantia da qualidade do *software*:** foco nas informações faltantes durante as várias fases do desenvolvimento dos aplicativos;

• **o processo de desenvolvimento do *software***: surgiu a definição de ciclo de vida no processo de desenvolvimento, dividindo-o em fases para maior garantia e controle das atividades e recursos envolvidos nos projetos.

2.2 IMPORTÂNCIA DAS MÉTRICAS

Definir indicadores para a medição é essencial para analisar a qualidade e produtividade do processo de desenvolvimento e também das manutenções em projetos de *softwares*. Pressman (1995), aponta alguns motivos para que o *software* seja medido:

- Indicar a qualidade do produto;
- Avaliar a produtividade das pessoas que produzem o produto;
- Avaliar os benefícios em termos de produtividade e qualidade derivados de novos métodos e ferramentas de *software*;
- Formar uma linha básica de estimativas;
- Ajudar a justificar os pedidos de novas ferramentas ou treinamento adicional;
- Entender e aperfeiçoar o processo de desenvolvimento;
- Melhorar a gerência de projetos e o relacionamento com clientes;
- Reduzir frustrações e pressões de cronograma;
- Gerenciar contratos de *software*;
- Indicar a qualidade de um produto de *software*;
- Avaliar a produtividade do processo;
- Avaliar os benefícios (em termos de produtividade e qualidade) de novos métodos e ferramentas de engenharia de *software*;
- Avaliar retorno de investimento.

Importante:

- Métricas devem ser facilmente calculadas, entendidas e testadas;
- Devem possibilitar estudos estatísticos;
- Devem ser expressas em alguma unidade de medida;
- Devem ser definidas tão logo se inicie o processo de desenvolvimento;
- Devem ser facilmente aplicadas em qualquer projeto, independente do observador;
- Devem ser válidas, confiáveis, práticas e de baixo custo.

Alguns conceitos importantes:

Medida

- Medida é uma função de mapeamento

Medição

- É o processo de aplicação da medida, em algo real.
- Para ser exata e confiável, a medição deve especificar:

  - Domínio: será medida a largura ou altura das pessoas?
  - Faixa: a medida da altura será feita em m ou cm?
  - Regras de mapeamento: será permitido medir altura considerando pessoas calçadas?

- No caso específico de desenvolvimento de *software*:

  - Tamanho do produto de *software* (ex.: número de Linhas de código);
  - Número de pessoas necessárias para implementar um caso de uso;
  - Número de defeitos encontrados por fase de desenvolvimento;
  - Esforço para a realização de uma tarefa;
  - Tempo para a realização de uma tarefa;
  - Custo para a realização de uma tarefa;
  - Grau de satisfação do cliente (ex.: adequação do produto ao propósito, conformidade do produto com a especificação).

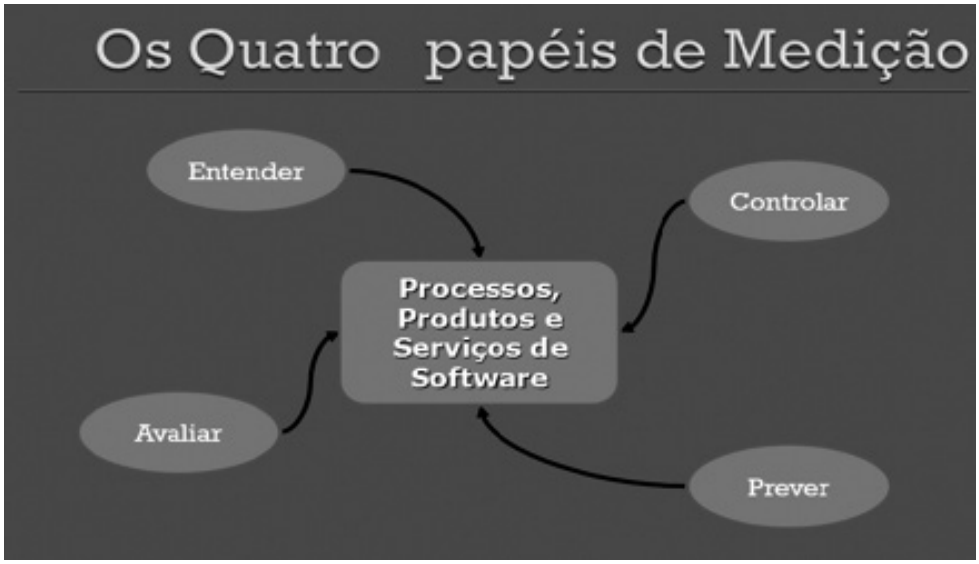
FIGURA 6 – EXEMPLOS DE MÉTRICAS

FONTE: Humphrey (1998, p. 43)

Papéis da Medição do *Software*:

Segundo Humphrey (1998), são quatro os principais papéis de Medições de *Software*:

FIGURA 7 – PAPÉIS DA MEDIÇÃO



FONTE: Humphrey (1998, p. 56)

- **Entender:** facilitam o entendimento dos processos envolvidos no desenvolvimento do *software.*
- **Avaliar:** o resultado auxilia na tomada de decisões em projetos.
- **Controlar:** facilitam o controle de todas as etapas do desenvolvimento do *software*.
- **Prever:** podem ser utilizadas para prever valores de atributos.

2.3 O PARADIGMA *GOAL QUESTION METRICS* (GQM)

É usado para definir o conjunto de métricas a ser coletado. Foi proposto por:

- Basili and Rombach's, *Goal-Question-Metrics Paradigm, IEEE Transactions on Software Engineering*, 1988.

Define que deve existir uma necessidade real para ser associada a cada métrica. Ou seja, cada métrica deve medir algo bem específico.

O processo tem início com a identificação dos interessados em efetuar a medição. Depois disso, são definidos os objetivos organizacionais do projeto e das tarefas que serão medidas. Definidos os objetivos são elaboradas as questões, cujas respostas serão apontadas de forma numérica pelas métricas escolhidas. No caso específico de projetos de *software*: qual o percentual de retrabalho do projeto? Qual o percentual de erros após a implantação? E, assim por diante...

Lembre-se:

É preciso definir as perguntas para então escolher as métricas. Nunca o contrário.

Dicas para selecionar métricas:

- Seja realista e prático.
- Considere o processo e o ambiente de desenvolvimento do cenário atual.
- Não selecione métricas difíceis de encaixar na realidade da medição.
- Comece devagar – com os dados que você tem.
- A equipe não deve se abalar com os processos de medição.

FONTE: Disponível em: <www.cin.ufpe.br/~if720/slides/introducao-a-metricas-de-software.ppt>. Acesso em: 29 fev. 2016.

O processo de medição é um processo dinâmico e cíclico que envolve planejamento, medição, análise de dados, tomada de decisões baseadas em análises de cenários, prática das decisões. Quando findada esta rotina, volta-se ao início e executa-se tudo novamente. Esse processo deve permitir a melhoria

contínua do processo, através da avaliação dos métodos e ferramentas utilizadas no processo de construção do projeto.

Importante:

- Medições devem ser usadas para medir processos, não pessoas.
- O processo de medição deve ter objetivos claros e bem definidos.
- O processo de medição deve ser fortemente acoplado com o processo de gerência da qualidade e integrado dentro de planos e orçamentos.
- O processo de coleta de dados deve ser simples, e ferramentas automáticas para extração de dados devem ser usadas.
- O processo de medição é contínuo e sujeito a melhoria.

FONTE: Disponível em: <http://www.bfpug.com.br/Artigos/Palestra%20_Medicoes%20Claudia%20Hazan.pdf>. Acesso em: 29 fev. 2016.

2.4 PARA PÔR EM PRÁTICA UM BOM PROGRAMA DE MEDIÇÃO

- Escolha um conjunto de métricas que melhor se adapte ao seu cenário de desenvolvimento de *software*.
- Procure associar as métricas às decisões que você precisa tomar.
- Avalie os processos. As pessoas devem ser avaliadas de outra forma, tendo como base os processos de Avaliação de Desempenho.
- Não use as métricas como forma de achar culpados para o que não deu certo.
- Não escolha nem defina as métricas sozinho. Envolva mais pessoas. De preferência, da equipe, e com conhecimento técnico.
- Estabeleça prioridades na medição.
- Integre o programa ao desenvolvimento de *software*, e alinhe-o aos objetivos estratégicos da organização.
- Use padrões e documente as medições.
- Compartilhe os resultados extraídos das métricas.
- Procure criar a cultura da medição dentro da organização, não restringindo a prática ao setor de TI.

2.5 SOBRE O PLANO DE MÉTRICAS

Para cada objetivo técnico o plano deverá conter as seguintes informações:

- POR QUE as métricas satisfazem o objetivo.
- QUE métricas serão coletadas, como elas serão definidas, e como serão analisadas.
- QUEM fará a coleta, quem fará a análise, e quem verá os resultados.
- COMO será feito: que ferramentas, técnicas e práticas serão usadas para apoiar a coleta e análise das métricas.
- QUANDO no processo e com que frequência as métricas serão coletadas e analisadas.
- ONDE os dados serão armazenados.

2.6 ESPECIFICANDO AS MEDIÇÕES – DEFINIÇÕES OPERACIONAIS

Para cada métrica devem ser definidos os objetivos e seu público-alvo. Questões importantes:

- Quem precisa da informação?
- Quem precisa das informações fornecidas pela métrica?
- Quando as informações devem ser coletadas? Com qual frequência?
- Quem vai coletar e armazenar as informações?
- Qual será a ferramenta de apoio para a coleta de dados?
- Onde a informação será armazenada e por quanto tempo estará disponível?
- Será guardado um histórico das métricas por projeto?
- O processo é automatizado o suficiente para que seja realmente eficaz?

2.7 SOBRE OS PROCEDIMENTOS DE ANÁLISE DAS MÉTRICAS

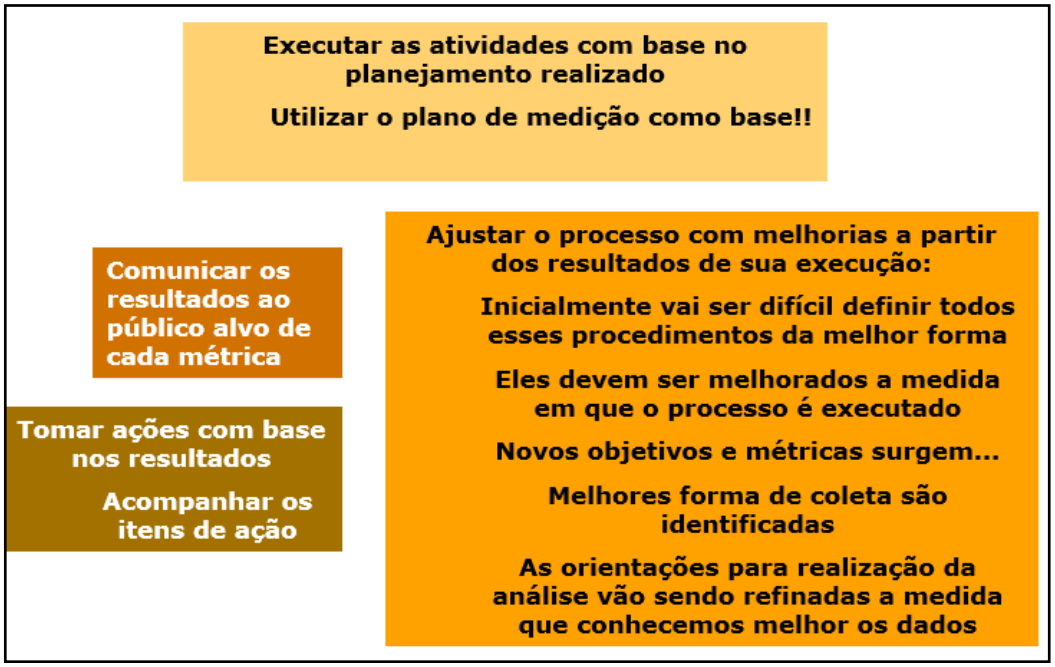
O processo de análise das métricas é importante, num primeiro momento, para que se compreenda a própria métrica utilizada e como a mesma pode ser aperfeiçoada, contribuindo assim para uma correta tomada de decisões acerca do bom andamento dos projetos.

- Em número;
- Em gráficos de barras, linhas, colunas, pizza, histogramas;
- Em diagramas;
- Em tabelas.

Devem ser estabelecidos os limites de controle, ou seja, definir até onde se deve medir, para não tornar o processo caro e burocrático demais.

Após todo o planejamento...

FIGURA 8 – SUGESTÕES PARA EXECUÇÃO APÓS O PLANEJAMENTO DA MEDIÇÃO



FONTE: Disponível em: <http://docslide.com.br/documents/2005-alexandre-vasconcelos-metricas-de-software>. Acesso em: 20 fev. 2015.

2.7.1 Sobre o armazenamento dos dados após a análise

Armazene todas as informações relevantes do processo: dados, resultados, decisões, problemas, riscos, benefícios.

Elabore uma política de acesso a estas informações, para evitar o uso indevido dos dados. Avalie as pessoas e seu comprometimento com o programa de medições. Procure fazer, sempre que possível, a comparação entre os projetos, para que seja possível melhorar o procedimento de medição adotado.

2.8 TENDÊNCIAS NO PROCESSO DE MEDIÇÃO

As tendências no processo de medição apontam para métodos que permitam avaliar a capacidade cognitiva, estrutural e funcional dos projetos; que possam ser extraídas nas etapas iniciais do mesmo, e não dependam de linguagens específicas para serem utilizadas no processo de medição.

3 MÉTRICAS TRADICIONAIS

Neste tópico serão abordadas algumas métricas tradicionais do processo de desenvolvimento de *software*. Por existirem muitas abordagens, somente as mais utilizadas serão relatadas aqui. Caso tenha interesse, consulte o autor Funck (1994).

3.1 ANÁLISE POR PONTOS DE FUNÇÃO (FPA)

A análise por pontos de função ainda é uma das técnicas mais utilizadas no desenvolvimento tradicional e estruturado. Ambler (1998) aponta os benefícios desta forma de medição:

- Medir o que o cliente/usuário requisitou e recebeu;
- Medir independentemente da tecnologia utilizada para implementação;
- Propiciar uma métrica de tamanho para apoiar análises da qualidade e produtividade;
- Propiciar um veículo para estimativa de *software*;
- Propiciar um fator de normalização para comparação entre *softwares*.

Esta métrica se baseia nas expectativas do usuário em relação ao sistema. Tem como unidade de medida os dados e as transações do sistema. É uma técnica útil para comparar sistemas construídos em ferramentas iguais, semelhantes ou distintas. Porém, pode ser considerado um processo caro, pois é um processo demorado, mesmo que o observador tenha experiência na extração e aplicação da técnica.

A maioria dos profissionais que adota este tipo de métrica, faz uso de uma planilha Excel para alimentar e armazenar as informações.

3.2 COCOMO (*CONSTRUCTIVE COST MODEL*)

COCOMO é o modelo construtivo de custos, calculado a partir do número de linhas do código-fonte do programa, que será entregue ao usuário (FUNCK, 1994). Com base no código de programação são definidos os prazos, recursos

humanos e técnicos e o custo final de desenvolvimento do projeto. Determinar o tamanho exato do *software* é uma das limitações deste método, pois não leva em conta a interface e outras variáveis envolvidas na construção do aplicativo.

3.3 LINHAS DE CÓDIGO (LOC - *LINES OF CODE*)

Esta é a técnica mais antiga e considera todas as linhas de código do programa. Os estudiosos divergem quanto ao fato de considerar ou não as linhas de comentários inseridos nos códigos-fonte. Outra situação é a recursividade (quando uma rotina chama a si própria – ex.: quando uma função de cálculo chama a si mesma no cálculo de uma média, por exemplo). A recursividade tende a deixar os códigos menos extensos, pois não há duplicidade dos mesmos.

Outra desvantagem apontada por Pressman (1995), é que esta técnica tem forte ligação com a tecnologia e linguagem de programação adotada no desenvolvimento. Isso impossibilita a comparação com projetos que utilizem linguagem de programação distintas. Outra desvantagem é que não permite estimar o tamanho do projeto na fase de levantamento de requisitos ou da modelagem.

Para Funck (1994) as vantagens de utilização desta técnica são:

- É uma técnica fácil de obter e utilizar.
- Muitos autores a referenciam, tornando abundante a literatura para pesquisa e posterior domínio.

O ideal é associar esta métrica com outras formas de medição, a fim de garantir a eficácia no levantamento do tamanho do projeto. Sozinha, não é significativamente confiável.

3.4 MÉTRICA DE CIÊNCIA DO *SOFTWARE*

Esta técnica surgiu em 1970, sendo pioneira na medição de código-fonte. Baseia-se no fato de que o *software* é um processo mental dos símbolos utilizados nos códigos dos programas. São considerados os verbos executáveis, como: IF, DIV, READ, ELSE e todos os operadores lógicos, e os operandos (variáveis e constantes). O tamanho é definido considerando o programa como sequência de operadores associados a operandos.

É uma técnica limitada pois também impossibilita a comparação entre projetos que utilizam diferentes linguagens, e, em consequência disso, não evoluiu, pois trata o sistema como um módulo.

3.5 MÉTRICA DA COMPLEXIDADE CICLOMÁTICA

Esta métrica mostra em forma de grafos a sequência de um programa em rotas diferentes. Para Shepperd (1995), os programas são representados por grafos dirigidos representando o fluxo de controle.

A partir de um determinado grafo, é extraída a complexidade ciclomática, que equivale ao número de decisões adicionais dentro de um programa. Pode ser calculada pela fórmula: **v(G) = E – n + 2**, onde, **E:** é o número de arestas e **N**: é o número de nós.

Ainda segundo Shepperd (1995), a complexidade ciclomática leva em conta o número de sub-rotinas dentro de um programa, sugerindo que as mesmas sejam tratadas como componentes não relacionadas dentro do grafo de controle.

Para calcular a complexidade ciclomática dos seus programas, acesse: <http://blog.caelum.com.br/medindo-a-complexidade-do-seu-codigo/#comments>.

RESUMO DO TÓPICO 1

Neste tópico você:

- Relembrou conceitos da Engenharia de *Software* acerca das métricas aplicadas no desenvolvimento de *software* tradicional e estruturado.
- Aprendeu a diferença entre métricas diretas e indiretas.
- Conheceu as métricas de tamanho, orientadas à função e orientadas a pessoas.
- Compreendeu a importância da adoção das métricas no processo de desenvolvimento dos aplicativos.
- Conheceu os quatro papéis da medição.
- Recebeu dicas para a seleção de métricas e para colocar em prática um bom plano de medição.
- Conheceu as definições operacionais do processo de medição.
- Conheceu os procedimentos a serem considerados na análise do resultado das métricas.
- Foi orientado quanto ao armazenamento dos dados do processo de medição: processo, procedimentos e resultados.

AUTOATIVIDADE

1 É um exemplo de métrica de controle de *software*:
a) O comprimento médio de identificadores em um programa.
b) O número de atributos e operações associadas com objetos em um projeto.
c) O tempo médio requerido para reparar defeitos relatados.
d) O número de mensagens de erro.

2 São objetivos das métricas de projeto:
a) Minimizar o cronograma e avaliar a qualidade do produto.
b) Avaliar a qualidade do produto e padronizar o projeto.
c) Padronizar o projeto e maximizar o lucro.
d) Minimizar intervenções do cliente e apontar padrões utilizados.

TÓPICO 2

MÉTRICAS PARA PROJETOS ORIENTADOS A OBJETOS

1 INTRODUÇÃO

A literatura de métricas para desenvolvimento de *software* Orientado a Objetos é farta, porém, carente nos aspectos que medem a qualidade. Isso se justifica pela subjetividade da questão e se percebe pela ausência de métricas precisas para este tipo de medição, pois indicadores para medidas de atributos bem como para a correção de defeitos ainda não foram devidamente estabelecidos.

2 MÉTRICAS PARA ORIENTAÇÃO A OBJETOS (OO)

O processo de medição de *software* OO é diferente do *software* tradicional, uma vez que este paradigma usa objetos e não algoritmos como termos de construção dos aplicativos, requerendo uma atenção diferente, tanto no projeto quanto em sua implementação.

A orientação a objetos pode ser considerada como a evolução ou mesmo a continuidade das técnicas de análise da programação estruturada. Os procedimentos estruturados concentravam-se em transformar as entradas em saídas. Ou seja, informações de cadastros e rotinas, transformavam-se em informações de relatórios e consultas, por exemplo. No procedimento orientado a objetos, o foco é o conteúdo das entidades, ou seja, os objetos.

> A análise e os projetos orientados a objeto, são continuidade da evolução das técnicas de análise estruturada. Esses métodos refletirão uma mudança na estratégia de análise, para incluir mecanismos fundamentais na elaboração de sistemas orientados a objeto. Num sistema orientado a objeto, a ênfase não é sobre a transformação de entradas em saídas, mas sobre o conteúdo das entidades, os objetos. Os sistemas orientados a objetos são mais flexíveis às mudanças. Quando uma mudança é necessária, a característica de hereditariedade permite a reutilização e extensão dos modelos existentes.

FONTE: Adaptado de: <http://campeche.inf.furb.br/tccs/1999-II/1999-2eduardojosecardosovf.pdf>. Acesso em: 1 mar. 2016.

Neste sentido, as métricas servirão para estipular e medir fatores como: recursos humanos / tempo de desenvolvimento, cronogramas, falhas, erros, retrabalhos. Além disso tudo, também auxiliam na definição do tamanho do projeto, sendo necessário iniciar a coleta de dados destas medições já na fase inicial dos projetos. Abaixo alguns exemplos de métricas relacionadas com o processo que podem ser coletadas quando se utiliza a Orientação a Objetos:

- quantidade total de classes;
- quantidade total de operações;
- quantidade de classes reutilizadas e número de classes desenvolvidas;
- quantidade de operações reutilizadas e número de operações desenvolvidas;
- quantidade de mensagens enviadas.

A experiência adquirida na medição de cada projeto permite definir e planejar melhor os projetos que virão. Tome como exemplo o seguinte: conhecendo o tempo médio gasto para especificar um caso de uso, poderíamos estipular o tempo para os demais casos. Mas lembre-se, por mais que se tenha experiência, cada projeto é um projeto. Eles se diferenciam pelos processos, pelas pessoas e pela cultura na qual estão sendo desenvolvidos (JACOBSON, 1992).

Estudaremos a seguir as métricas mais utilizadas no desenvolvimento Orientado a Objetos. Elas estão divididas em três categorias distintas: métricas de análise, métricas de projeto e métricas de construção.

2.1 MÉTRICAS DE ANÁLISE

As técnicas de análise são utilizadas para medir todos os recursos necessários ao desenvolvimento desta atividade.

2.1.1 Porcentagem de classes-chave

Classes-chave são as classes mais importantes e estão diretamente relacionadas ao negócio. São utilizadas para determinar o tamanho do projeto e indicam o final do processo de análise. É um processo simples de medir, pois depende diretamente da construção dos diagramas. Examinando o diagrama de classes, o percentual deve ficar entre 30% e 50%. Se ficar abaixo, significa que o processo de análise não está concluído (LORENZ; KIDD, 1994).

2.1.2 Números de cenários de utilização

Considera como indicador de medição, os cenários de utilização na visão do usuário, sendo que os pequenos aplicativos possuem entre 5 e 10

cenários. Aplicativos de média complexidade têm entre 20 e 30 cenários. Os desenvolvimentos mais críticos envolvem mais de 40 cenários (AMBLER, 1998).

2.2 MÉTRICAS DE PROJETO

Permitem comparar planos de medições de projetos distintos e propõem recomendações para projetos futuros.

2.2.1 Contagem de métodos

Para Ambler (1998), as classes com menos métodos possuem uma tendência a serem mais reutilizáveis. Faz-se então a contagem dos métodos de todas as classes da aplicação. As classes com mais métodos são analisadas para averiguar se podem ser aplicadas em outros projetos. Surge então o esforço para torná-la o mais reutilizável possível. Os números de métodos de uma classe ficam no intervalo de 20 a 40.

2.2.2 Métodos ponderados por classe (*WMC - Weigthed Methods per Class*)

Esta é a forma de medir individualmente uma classe. Contam-se os métodos e somam-se as suas complexidades ciclomáticas. O número de métodos e sua complexidade são indicadores de quanto tempo e esforço serão necessários para desenvolver e manter a classe. Quanto maior o número de métodos da classe, maior será o impacto nos filhos da mesma, por causa dos métodos herdados da classe de origem.

2.2.3 Resposta de uma classe (RFC – *Response For a Class*)

RFC é composta pelo número de métodos distintos chamados em resposta a uma mensagem por um objeto ou método da classe, combinando a complexidade da mesma com a comunicação feita com outras classes. Quanto maior é o número de métodos chamados em resposta a uma determinada mensagem, mais complicado será o seu teste, pelo fato de demandar maior assimilação da situação. Para Rosenberg (1998), um número aceitável para esta métrica seria 100, entretanto, para a maioria das classes este número é menor ou igual a 50.

2.2.4 Profundidade da árvore de herança (DIT - *Depth of Inheritance Tree*)

A profundidade da hierarquia da herança é o número máximo de passos da classe nó até a raiz da árvore e é medida pelo número de classes ancestrais (ROSENBERG, 1998). Segundo Ambler (1998), esta métrica indica as dificuldades na maneira de utilizar o conceito de herança. Árvores muito profundas constituem projetos de maior complexidade, uma vez que um número maior de métodos e classes está envolvido, pois quanto mais profunda uma classe na hierarquia, maior o número de métodos que ela provavelmente herda, tornando-a mais difícil de entender e, portanto, de manter e incrementar.

Segundo Rosenberg (1998), DIT igual a zero indica uma classe raiz, um alto percentual de DIT entre dois e três indica um alto grau de reutilização, entretanto se a maioria dos ramos da árvore forem pouco profundos (DIT < 2) isto pode representar uma pobre exploração das vantagens do desenvolvimento OO e da herança. Por outro lado, uma árvore muito profunda (DIT > 5) pode ser perigosa, pois aumenta a complexidade do projeto. Para Ambler (1998), se a profundidade de uma herança for maior do que cinco, é preciso reavaliar o projeto.

2.2.5 Número de filhos (NOC – *Number Of Children*)

É o número de subclasses subordinadas a uma classe. Está diretamente relacionada com a profundidade da árvore de herança. O número de filhos indica a influência da classe no projeto como um todo. Quanto mais filhos, maior é a reutilização.

2.2.6 Falta de coesão (LCOM - *Lack Of Cohesion*)

De acordo com Rosenberg (1998), a falta de coesão é definida pelo número de diferentes métodos dentro de uma classe que referenciam uma determinada variável de instância. A falta de coesão mede as diferenças de métodos em uma classe pelos atributos ou variáveis de instância.

Segundo Rosenberg (1998), existem pelo menos duas maneiras de medir a coesão:

a) calcular para cada atributo de uma classe o percentual de métodos que o utilizam, obter a média destes percentuais e subtraí-la de 100%. Percentuais baixos significam uma maior coesão entre atributos e métodos na classe;
b) outra maneira é contar o número de conjuntos disjuntos produzidos pela intersecção dos conjuntos de atributos de diferentes classes.

2.2.7 Acoplamento entre objetos (CBO – *Coupling Between Object Classes)*

O acoplamento é medido pelo número de outras classes acopladas (conectadas) a uma classe específica.

De acordo com Rosenberg (1998), as classes (objetos) podem ser acopladas de três maneiras:

a) quando uma mensagem é passada entre os objetos;

b) quando os métodos declarados em uma classe utilizam atributos ou métodos de outras classes;

c) através da herança que introduz um significativo acoplamento entre as superclasses e suas subclasses.

Esta medição é útil para avaliar a complexidade dos testes dos vários módulos do sistema. Quanto maior é o acoplamento, mais criterioso será o teste.

2.2.8 Utilização Global

Esta métrica conta as variáveis globais utilizadas na aplicação. Quanto maior o número de variáveis globais, maior será o acoplamento e mais rigorosos os testes. É melhor, portanto, optar por reduzir a sua utilização.

2.3 Métricas de Construção

São métricas que permitem aferir e melhorar a qualidade do projeto. Abaixo são listadas algumas métricas de construção.

2.3.1 Tamanho do Método

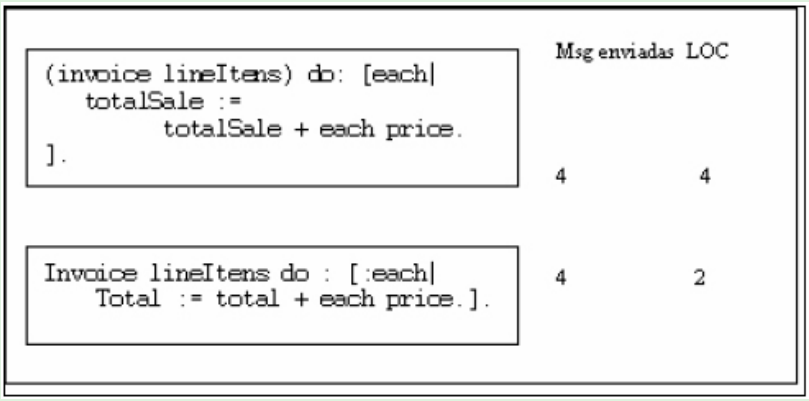
2.3.1.1 Quantidade de mensagens enviadas

É a contagem do total de mensagens enviadas por um método. É considerada uma métrica imparcial.

A linguagem utilizada influencia nesta métrica, pois quando se codifica numa linguagem híbrida, como C++, pode-se escrever métodos que não são orientados a objeto. Este código não seria contado no número de mensagens enviadas, mas relaciona-se com o tamanho do método. Por exemplo, não se pode ignorar 100 linhas de código não orientado a objetos e contar somente um par de mensagens enviadas.

A figura a seguir mostra dois pedaços de código *Smalltalk* que fazem exatamente a mesma coisa, porém com estilos de codificação diferentes. Se estes dois pedaços de código forem comparados baseando-se na contagem de linhas físicas o primeiro caso teria um valor maior. Contando as mensagens a comparação entre os dois é mais exata.

FIGURA 9 – EXEMPLO DE CÓDIGO *SMALLTALK*



FONTE: Lorenz (1994)

FONTE: Disponível em: <http://dsc.inf.furb.br/arquivos/tccs/monografias/20012patriciareginaramosdasilvaseibtvf.pdf>. Acesso em: 1 mar. 2016.

2.3.1.2 Linhas de código (LOC)

Conta número de linhas de código ativas em um método, não considerando o estilo da codificação. Métodos menores são mais fáceis de executar a manutenção. Entende-se que quando o método é maior ele é orientado a função e não segue o conceito da orientação a objetos. Os objetos precisam de outros objetos para executar as atividades. Isso faz com que a construção dos métodos não se estenda demais. Logo, o estilo do programador em escrever o código, influenciará diretamente esta métrica, pois quebra uma mensagem em diferentes linhas físicas, gerando um aumento na contagem de linhas de código. Para Ambler (1998), os métodos devem ser pequenos, porém bem comentados, e devem ter um objetivo específico.

2.3.1.3 Média do tamanho dos métodos

O tamanho médio dos métodos pode ser obtido pela média de linhas de código por métodos (total de linhas de código dividido pelo total de métodos) ou pela média de mensagens enviadas (total de mensagens enviadas dividido pelo total de métodos).

FONTE: Disponível em: <http://bit.ly/2W4iTTt>. Acesso em: 1 mar. 2016.

2.3.2 Percentual comentado

Considera as linhas comentadas por métodos. Poucos comentários indicam mais dificuldade de outro programador interpretar o método. O contrário pode indicar um tempo maior e desnecessário na documentação do mesmo.

Obtém-se este percentual através da divisão do total de linhas comentadas pelo total de linhas de código. Para Rosenberg (1998), este percentual deveria ficar entre 20% e 30%, pois os comentários auxiliam na manutenção e reusabilidade do código.

2.3.3 Complexidade do método

Geralmente consideram as decisões implementadas no código que são representadas por comandos ***if-then-else*** e outras. As medições mais tradicionais não consideram as diferenças dos projetos OO.

Segundo Lorenz (1994), o valor esperado para esta métrica é 65. Este valor originou-se da avaliação de um número significativo de projetos ao longo dos anos e julgamento baseado na experiência do próprio autor. A média da complexidade por método é obtida pelo número de complexidades dividido pelo número de métodos.

FONTE: Disponível em: <http://dsc.inf.furb.br/arquivos/tccs/monografias/20012patriciareginaramosdasilvaseibtvf.pdf>. Acesso em: 1 mar. 2016.

2.3.4 Tamanho da classe

O tamanho da classe pode ser medido de várias formas. A seguir, listamos algumas para seu maior entendimento.

2.3.4.1 Quantidade de métodos de instância públicos em uma classe

Por este método mede-se a responsabilidade que a classe tem dentro do sistema. Métodos públicos são indicadores do trabalho total feito por uma classe, uma vez que eles são serviços utilizados por outras classes. A forma de obtenção é a contagem da quantidade.

2.3.4.2 Quantidade de métodos de instância em uma classe

É a contagem de todos os métodos de instância de uma classe. O número está relacionado ao total de colaboração utilizada pela classe. Entende-se que uma classe maior é mais difícil de manter. As classes menores são mais reutilizáveis. Um indicativo de maior responsabilidade da classe é o fato dela apresentar muitos métodos, mesmo excluindo-se os herdados.

O ideal é que haja a distribuição adequada de trabalho entre as classes. Segundo Lorenz (1994), o valor recomendado para esta métrica é de 40 para classes de interface com o usuário e 20 para as demais. Através desta métrica pode-se obter outra que é a média de métodos de instância por classe. Esta média é obtida pelo total de métodos instanciados dividido pelo total de métodos.

2.3.4.3 Quantidade de atributos por classe

A quantidade de atributos por classe possibilita definir um indicador para a qualidade dos projetos orientados a objetos. Um número elevado de atributos indica um maior número de relacionamentos com outros objetos, no mesmo sistema. Segundo Ambler (1998), as classes com mais de três ou quatro atributos estão com frequência mascarando um problema de acoplamento na aplicação. Conforme Lorenz (1994), para classes de interface com o usuário este número pode chegar a nove, pois estas classes necessitam de mais atributos para lidar com componentes de telas.

2.3.4.4 Média de atributos por classe

A média de atributos indica o tamanho da classe. Chega-se a ela dividindo-se o total de atributos pelo total de métodos. Muitos atributos indicam maior trabalho e responsabilidade para uma classe específica, sendo que esta pode estar tendo mais relacionamentos com outros objetos do que deveria.

2.3.4.5 Quantidade de métodos de classe em uma classe

Esta métrica cria um indicador do total de métodos que são usados por todas as instâncias. Indica carência em projetos, caso houver maior manipulação de serviços por instâncias individuais, do que pela classe em si.

Lorenz (1994) aponta dois tipos de entradas para esta métrica:

- **absoluto**: as classes raramente necessitam mais do que quatro métodos de classe;
- **relativo a quantidade de métodos de instância**: este valor é obtido com a combinação da entrada desta métrica com a entrada esperada da quantidade de métodos de instância (quatro *versus* vinte), o que resulta em 20%, este valor é utilizado para identificar possíveis anormalidades.

2.3.4.6 Quantidade de variáveis de classe em uma classe

As variáveis de classe são globais e utilizadas para armazenar valores constantes e customizáveis que afetam a funcionalidade das instâncias Lorenz (1994).

Esta métrica é obtida dividindo-se o total de variáveis de classe pelo total de classes.

Lembre-se que algumas classes têm mais variáveis de classe, mas mesmo assim, devem ter menos variáveis do que atributos. Ainda, segundo o autor, as classes têm em média três variáveis de classe.

2.3.5 Quantidade de classes abstratas

A utilidade das classes abstratas é facilitar a reutilização de métodos e classes entre as suas subclasses. A quantidade extraída de classes abstratas é um forte indício da herança e esforços empenhados na resolução de um problema. Um projeto bem definido possui em média mais de 10% de classes abstratas.

2.3.6 Uso de herança múltipla

A linguagem C++ é um tipo de linguagem que permite que uma classe herde funcionalidades de uma superclasse. Porém podem ocorrer alguns problemas como conflitos dos nomes e entendimento por parte de quem irá codificar o desenvolvimento. Alguns autores concordam que a herança múltipla não é necessária, não sendo recomendado o seu uso.

2.3.7 Quantidade de métodos sobrescritos por uma subclasse

> Uma subclasse pode definir um método com o mesmo nome de sua superclasse. Isto é chamado de sobrescrever o método, pois uma mensagem agora irá causar a execução do novo método, em vez do método da superclasse. Muitos métodos sobrescritos indicam um problema de projeto, pois as subclasses deveriam ser uma especialização de suas superclasses, elas deveriam ampliar os serviços das superclasses. A média de métodos sobrescritos por classe pode ser obtida pelo total de métodos sobrescritos dividido pelo total de classes. (LORENZ, 1994, p. 61)

O autor aponta ainda que o número ideal de métodos sobrescritos por uma subclasse deve ficar em torno de três, sendo que este número deve ser menor nos níveis mais baixos da árvore de herança.

2.3.8 Quantidade de métodos herdados por uma subclasse

A entrada desta métrica é o inverso da métrica quantidade de métodos sobrescritos, e um percentual baixo indica subclasses pobres (LORENZ, 1994).

2.3.9 Quantidade de métodos adicionados por uma subclasse

As subclasses dependem de um método para justificar sua existência e função. Menor é a quantidade de métodos conforme se desce a hierarquia na árvore de herança. Para classes no nível mais baixo, o número de métodos fica em torno de 20. No nível cinco, por exemplo, estaria em torno de quatro métodos apenas.

2.3.10 Índice de especialização

Podemos entender por especialização a capacidade de se adicionar mais funcionalidades em uma subclasse que já existe, como forma de complemento das funções da mesma.

Para que isso seja possível é necessário:

- Adicionar novos métodos;
- Adicionar funções a métodos existentes, embora ainda chame o método da superclasse;
- Sobrescrever métodos com uma função totalmente nova;
- Excluir métodos através da sobrescrita dos mesmos por outros sem funções.

A fórmula desta métrica é: (número de métodos sobrescritos * nível da classe na árvore de herança) / total de métodos. O sinal de anormalidade é indicado por um percentual igual ou superior a 15%.

FONTE: Disponível em: <http://dsc.inf.furb.br/arquivos/tccs/monografias/20012patriciareginaramosdasilvaseibtvf.pdf>. Acesso em: 1 mar. 2016.

RESUMO DO TÓPICO 2

Neste tópico você aprendeu que:

- O processo de medição de *software* OO é diferente do *software* tradicional, uma vez que este paradigma usa objetos e não algoritmos como termos de construção dos aplicativos, requerendo uma atenção diferente, tanto no projeto quanto em sua implementação.

- Conheceu as métricas de análise para projetos orientados a objetos.

- Conheceu as métricas de projeto para projetos orientados a objetos.

- Conheceu as métricas de construção para projetos orientados a objetos.

- Adquiriu conhecimento para identificar e selecionar o conjunto de métricas mais adequado ao seu projeto orientado a objetos.

AUTOATIVIDADE

1 O gerente solicitou que fossem colhidas algumas métricas de performance do processo de fabricação de um carro, por exemplo. Dentre as métricas selecionadas, qual NÃO constitui um índice de performance aplicável neste contexto, já que sua "otimização" poderia afetar a qualidade do processo?
a) Quantidade de peças substituídas por revisão.
b) Tempo por revisão.
c) Tempo por atividade.
d) Quantidade de revisões por dia

2 Cite as três categorias de métricas de software presentes na literatura e liste as principais técnicas de medição para cada uma delas.

TÓPICO 3

PADRÕES DE PROJETO: CARACTERÍSTICAS E TIPOS

1 INTRODUÇÃO

Os padrões de projeto existem para representar as soluções de problemas reais resolvidos através de implementações orientadas a objetos. Os padrões identificam classes de objetos e seus relacionamentos, bem como suas funcionalidades e responsabilidades. Existem vários tipos de padrões de projetos e cada um deles busca resolver uma situação específica.

2 DESCRIÇÃO DE UM PADRÃO DE PROJETO

Um padrão de projeto geralmente define um nome, o problema e sua respectiva solução em orientação a objetos. Descrever o padrão utilizado é importante para delimitar suas características e compará-las com outros padrões. Isso facilita na hora de escolher o padrão mais apropriado para o desenvolvimento do projeto.

Gamma (2000, p. 63) divide os padrões nas seguintes seções:

> • **Nome do padrão e classificação:** deve indicar o objetivo do padrão, de forma clara.
> • **Intenção do padrão:** descreve o que o padrão faz; qual a sua funcionalidade específica; o porquê de ele existir.
> • **Também conhecido como:** outro nome ou apelido, caso exista.
> • **Motivação:** descreve um problema no projeto e como o padrão auxilia e favorece a sua solução.
> • **Aplicabilidade:** onde e como o padrão pode ser aplicado.
> • **Estrutura:** representa graficamente as classes no padrão, usando a UML para representar os relacionamentos entre os objetos.
> • **Participantes:** apresenta as classes e objetos que fazem parte do padrão e quais as suas responsabilidades.
> • **Colaborações**: como os participantes colaboram para atender as suas responsabilidades.
> • **Consequências:** descreve como o padrão alcança os objetivos anteriormente traçados.
> • **Implementação:** detalhes que devem ser observados, como a linguagem escolhida para a customização, por exemplo.
> • **Exemplo de código:** um exemplo de código que demonstre como o padrão deve ser implementado em alguma linguagem.
> • **Usos conhecidos:** exemplos de padrões já utilizados em outros sistemas.

• **Padrões relacionados:** apresenta padrões que se assemelham a este padrão e quais as diferenças. Apresenta também outros padrões que podem ser utilizados em conjunto.

3 CLASSIFICAÇÃO DOS PADRÕES DE PROJETOS

Existe um grande número de padrões disponíveis na literatura atualmente. Para facilitar o estudo e respectiva escolha, os mesmos foram agrupados em categorias diferentes: padrões de criação, padrões estruturais e padrões comportamentais.

Padrões de criação: são utilizados na etapa de criação dos objetos e, de acordo com o escopo, se referem à criação de objetos para subclasses (escopo de classe) ou para outros objetos (escopo de objeto).

Padrões estruturais: se preocupam com a composição das classes e dos objetos. Dependendo do escopo se utilizam de herança entre classes para realizar a composição (escopo de classe) ou apresentam formas de como agregar objetos (escopo de objeto).

Padrões comportamentais: tratam a forma como classes e objetos, interagem e distribuem responsabilidade.

4 PADRÕES DE ANÁLISE

Podemos entender como padrões de análise, os modelos de negócio.

4.1 O QUE SÃO MODELOS

Os modelos são criados para tornar mais fácil o entendimento dos problemas. Vários modelos podem ser usados para representar um mesmo problema. Logo, podemos entender que não existe um modelo ou padrão mais correto, mas sim, um modelo/padrão mais adequado para cada situação. Escolher um modelo inadequado pode colocar em risco o projeto, dificultando a sua manutebilidade, flexibilidade e reusabilidade. Opte sempre por escolher modelos mais simples e práticos.

Observe sempre a independência do modelo em relação à tecnologia utilizada no desenvolvimento. Quanto mais independente da tecnologia melhor, pois poderão ser reutilizados em projetos distintos.

4.2 *ARCHETYPE PATTERNS*

Archetype Patterns são padrões que descrevem o alicerce das estruturas de negócio. Usam um nível de abstração maior do que os padrões de análise. São mais generalistas.

4.2.1 O que é *Archetype*

Podemos entender a palavra como "padrão original". Pelo dicionário Michaelis: padrão ou modelo. Tem origem na palavra grega ***archetypo.*** É algo importante e que ocorre com determinada frequência e de forma consistente.

Define o alicerce de um modelo, e mesmo que ocorram alterações, a ideia ou estrutura essencial é preservada. Com isso é possível fazer a modelagem de conceitos de negócios no intuito de reutilização em várias situações, pois podem ser adaptados sem o risco de perda do conceito original para o qual foi escolhido e criado.

"Um *archetype pattern* de negócio é uma colaboração entre *archetypes* de negócios que ocorrem consistente e universalmente em ambientes de negócios e sistemas de *softwares*." (MEDEIROS, 2000, p. 65).

Algumas características essenciais são definidas para *archetypes* e *archetype patterns*, como:

- **Universal**: deve ser aplicado de maneira uniforme e consistente em cenários distintos ao se considerarem domínios de negócios e aplicativos.
- **Difundido**: um único comportamento definido, tanto para domínio de negócios quanto para sistemas.
- **Possuir um histórico**: deve ter um conceito bem elaborado e que já seja conhecido.
- **Evidente para especialistas**: deve ser amplamente claro para aqueles profissionais experientes no seu domínio. Se não for de claro e rápido entendimento, não pode ser apresentado como um *archetype*.

4.2.2 *Archetype Patterns* e Padrões de Análise

Archetype Patterns e Padrões de análise, são conceitos distintos. *Archetypes* buscam identificar os conceitos considerados universais em sua aplicação. Os padrões de análise, porém, não se preocupam com isso, pois estão em um nível de abstração inferior.

As classes ou padrões de análise descrevem situações reais, em cenários reais. Simplesmente existem para agilizar o entendimento do problema ou do negócio de forma geral e abrangente. Existem também as classes de projetos que descrevem o nível mais técnico da situação, mesmo assim com abstração inferior aos *Archetype Patterns.*

4.2.3 Características dos *Archetype Patterns*

Archetype Patterns têm como principal característica o princípio da variação, que nada mais é do que a capacidade de adaptação para representar a mesma situação de formas distintas.

São três os tipos de variações impostas aos ***archetype patterns***:

- **Variação de *archetype***: um *archetype* pode possuir diferentes recursos (atributos, operações, constantes) para ser eficiente em diferentes contextos.

- **Variação de *archetype pattern***: alguns recursos são opcionais no padrão, de maneira que podem ser omitidos se forem desnecessários para o domínio em questão.

- ***Pleomorphism*:** é um tipo especial de variação de *archetype patterns*, neste caso o padrão pode assumir diferentes estruturas (diferentes *archetypes*, recursos dos *archetypes* e relacionamentos) para se adaptar às necessidades de um contexto de negócio.

4.2.4 Descrições de Padrões de Projeto

A seguir são listados os tipos de padrões de projeto!

4.2.4.1 *Abstract Data Type* (Classe)

O propósito é ocultar a estrutura de dados e acessar algoritmos entre mudanças não sensitivas de interface; com flexibilidade para mudar / repor implementação, I / O, sistema operacional, recursos de memória, sem afetar os clientes. Sua implementação está em fazer a interface independente de prováveis mudanças.

É possível criar múltiplas instâncias de ADT, mas somente um módulo e uma instância podem combinar vários relacionamentos de ADTs. Alguns exemplos são: Ø *Repository* (Base de Dados) onde o propósito é prover uma estrutura de dados central com interface de acesso para múltiplos clientes.

Clientes estes que são independentes, tendo, assim, a flexibilidade para adição e remoção dos mesmos e a implementação da estrutura de dados. Pode receber, estabelecer, questionar e atualizar métodos. Ø Cliente Servidor com depósito distribuído, onde clientes e servidores rodam em computadores diferentes, conectados por alguma rede.

Gerenciador (Coleção) com funções semelhantes a criar / deletar, registrar, procurar, *layout* e *display* dentro da classe, separadas de objetos na coleção. Pode usar *Singleton* para o componente gerenciador.

4.2.4.2 *Abstract Factory*

Seu objetivo é prover uma interface para criação de famílias ou objetos dependentes, sem especificar suas classes concretas e empacotar construtores para a família de objetos relatados dentro de um objeto; sendo mais apropriado quando o número e tipos de objetos ficam constantes. Sua implementação está na interface comum do construtor através de famílias e múltiplas implementações.

4.2.4.3 *Adapter*

Converte a interface da classe com outra interface cliente, permitindo que as classes trabalhem juntas. Seu propósito consiste na conversão de uma determinada interface dentro de outra, adaptando-as.

4.2.4.4 *Blackboard*

Tem como propósito decidir dinamicamente que transformadores (conhecidos como origens) se aplicam na distribuição de estrutura de dados; podendo adicionar/repor transformadores e repor controlador. Tem os seguintes componentes: distribuição da estrutura de dados, conjunto de transformadores e controladores que selecionam transformadores.

4.2.4.5 *Bridge*

Desliga a união de sua implementação, permitindo que a abstração e implementação se desenvolvam separadamente, podendo adicionar novas abstrações ou implementação, sem afetar outras. Possibilita a separação de hierarquias de classes, para que estas trabalhem juntas e se desenvolvam independentes; a manutenção e o aumento das abstrações lógicas, sem tocar na dependência do código, e vice-versa. Sua implementação consiste em separar camadas para abstração e implementação, além de fixar interface de implementação.

4.2.4.6 *Broker*

O propósito é encontrar servidores para requisições de clientes, adicionando/removendo servidores e clientes transparentemente através do protocolo para registrar servidores e serviços.

4.2.4.7 *Builder*

Separa a construção de um objeto complexo de sua representação, porém, alguns processos de construção podem ter diferentes representações. Especifica a criação do esqueleto usando construtores primitivos e usa a Delegação ao invés de *subclassing* para construção de primitivas.

4.2.4.8 *Bureaucracy*

Organizar uma estrutura hierárquica de objetos semelhantes que mantém a consistência interna, podendo estender a hierarquia e estender e transferir as responsabilidades. Sua implementação está na combinação de *Composite*, *Observer* e medida de responsabilidade.

4.2.4.9 Responsabilidade da Cadeia

Com o propósito de passar uma requisição para baixo da medida de objetos, evita o acoplamento do remetente da requisição para o destinatário. Sua flexibilidade está no desligamento e reconhecimento de uma requisição não conhecida *a priori* (determinada dinamicamente); na extensibilidade e na adição de novos reconhecimentos. Possui interface comum para reconhecedores.

4.2.4.10 Chamada a *Procedure* Remoto

Gerenciamento de Processo. Possui uma região condicional crítica (Mutex) chamada Semáforo. Esta região tem duplo fechamento checado; monitor; evento *loop*, *buffer* ilimitado, objeto ativo (futuro), *token* assíncrono, *half-sync* / *half-async*, transação e *rollback*, distribuição; resolvendo problemas que aparecem em sistemas distribuídos com recursos distribuídos.

4.2.4.11 *Command*

Encapsula a requisição como um objeto, através da permissão de clientes parametrizados com diferentes requisições e determina uma interface uniforme para requisições que permitam a configuração de clientes; pode delegar todo, parte ou nenhuma das requisições de implementação para outros objetos; discute os mecanismos *undo* e *redo* construídos em interfaces e adiciona processador de comando. Possui flexibilidade para múltiplos comandos, adiciona funcionalidades como *undo* / *redo*, *scheduling*.

4.2.4.12 *Composite*

Tem o propósito de reconhecer partes / todo das hierarquias; clientes consideram múltiplos objetos atômicos e *composites* uniformemente. Possui, também, interface comum para *composites* e átomos em superclasse e múltiplas implementações.

4.2.4.13 Concorrência

Esta categoria resolve problemas em processamento paralelo.

4.2.4.14 Controle

Distribui com controles a execução e seleção de métodos certos, no tempo certo. É subdividido em Controle de Processo que tem como objetivo regular o processo físico, ajustar / repor o controlador. É implementado em controlador, processos variáveis, variáveis de entrada e manipuladas e sensores. Suas variações são:

- *Open-loop-system* (processos variáveis não usados para ajustar o sistema).
- *Closed-loop-system*
- *Feedback control* (variáveis controladoras usadas para ajustar o sistema).
- *Feedforward control* (variável de entrada ou intermediária usada para ajustar o sistema).

4.2.4.15 *Convenience Patterns*

Simplificam o código. São subdivididos em Método *Convenience* que simplifica o método pela redução do número de parâmetros; define métodos especializados, chamados por métodos gerais, fornecendo, assim, combinações de parâmetros.

4.2.4.16 *Data Management*

Empacota o estado dos objetos genericamente, independente do conteúdo atual dos objetos.

4.2.4.17 *Decorator*

Anexa responsabilidades adicionais para um objeto dinamicamente; fornece uma alternativa flexível, estendendo funcionalidades, além de capturar o relacionamento classe-objeto que suporta o transparente "embelezamento". O termo "embelezamento" refere-se a quaisquer responsabilidades para um objeto, como por exemplo uma árvore com sintaxe abstrata e ações semânticas, um estado finito autômato com novas transações, ou uma rede de objetos persistentes com *tags*.

4.2.4.18 *Decoupling*

O propósito deste padrão é dividir o *software* em partes, de maneira que as partes individuais possam ser construídas, mudadas e reusadas independentemente; sua vantagem é a mudança local através da modificação de uma, ou somente algumas partes do sistema, ao invés do todo. Módulo, Abstração, *Data Type* e Camadas Hierárquicas têm larga aplicabilidade. *Iterator*, *Proxy*, *Facet e Visitor* se aplicam em situações de projeto restritas.

4.2.4.19 Estado

Permite que um objeto altere seu comportamento, quando seu estado interno mudar. Seu propósito é escolher o comportamento de acordo com o estado do objeto, adicionando/removendo estados. Pode ser implementado por formulários distribuídos para o estado finito e interface uniforme para ações. Implementa ainda cada estado do objeto e comportamento apropriado para um estado determinado.

4.2.4.20 Evento Baseado na Integração

Permite que os objetos se comuniquem indiretamente, sem o conhecimento do outro, integrando, assim, objetos que não se conhecem. É implementado para registrar participantes em canais comuns que enviam eventos/dados para o canal.

4.2.4.21 *Facade*

Provê uma interface unificada para o conjunto de *interfaces*, no subsistema, ocultando alguns componentes. Pode mudar/repor subsistemas ocultados através da inclusão de *interfaces*.

4.2.4.22 *Facet*

Adiciona novas interfaces para classes existentes, sem mudá-las; provendo múltiplas *views*. Com flexibilidade para interface e funcionalidade estendidas, implementa registros de um objeto e os retorna, se questionado.

4.2.4.23 *Flyweight*

Usa a distribuição para suportar um número largo de objetos de grãos finos eficientemente, salvando espaço pela distribuição de estados

comuns entre objetos. Sua implementação está em diferenciar entre estados intrínsecos e extrínsecos.

4.2.4.24 *Framework*

Conjunto de classes cooperando para fazer o projeto reutilizável para uma classe específica de *software* através da redefinição dos parâmetros do projeto, da captura de decisões do projeto e inclusão de subclasses concretas. Pode ser gerado para construir editores gráficos, para diferentes domínios ou compiladores para diferentes linguagens de programação; ou ainda, para uma aplicação particular.

Assim, o *framework* oferece um mecanismo para que o gerente de projetos identifique quais atributos do *software* correspondentes à sua corretude desempenho funcionais são importantes e um meio para avaliar o progresso no processo de desenvolvimento em relação às metas de qualidade. Seu propósito é prover uma camada da aplicação completa que poderá ser estendida pela subclasse, implementando subclasses, métodos; métodos *factory; builders e factories* abstratos.

4.2.4.25 Gerenciamento da Variável

Objetos diferentes podem ser considerados uniformemente em algum programa.

4.2.4.26 Integração

Permite o desenvolvimento independente de componentes que trabalham juntos.

4.2.4.27 *Iterator*

Captura as técnicas para acesso e passagem de estruturas dos objetos; ilustra como o encapsulamento da concepção das variáveis ajuda na flexibilidade e reusabilidade; além de fornecer uma maneira para acessar os elementos de um objeto agregado sequencialmente, sem expor a representação básica e proporcionar uma interface acessando componentes em um *aggregate* / *container*.

4.2.4.28 Máquinas Virtuais

Uma máquina virtual executa programas escritos em uma linguagem específica. Alguns exemplos são descritos a seguir:

- O Interpretador usa a representação para interpretar sentenças na linguagem, depositando componentes do programa e trabalhando com a memória, contador do programa e métodos para executar instruções na linguagem.
- O Emulador simula um processador de *hardware* no *software*, pois modificar o emulador é mais fácil que mudar o *hardware*.
- A Regra Baseada no Interpretador interpreta o conjunto regra, usando o fato base, com flexibilidade para repor o interpretador. Implementa um depósito de regras dos componentes, trabalhando com a memória, competidor de regra (ao invés do contador do programa) e a regra do interpretador.

4.2.4.29 *Mediator*

Define como um conjunto de objetos encapsulados se interagem; desprende o acoplamento pela manutenção de objetos explicitamente. Sua implementação se resume em separar classes para objetos e *mediator*; objetos informam *mediator* de eventos significantes pela chamada direta ou através da notificação do evento.

4.2.4.30 Memento

Sem a violação do encapsulamento, captura e exterioriza o estado do objeto interno, salvando e restaurando-o. Sua implementação está em empacotar / desempacotar rotinas para exteriorizar / restaurar o estado.

4.2.4.31 Mestre / Escravo

Dinamicamente distribuídos, eles trabalham lado a lado com processos subordinados ao protocolo, para o trabalho distribuído; podendo adicionar / remover escravo e escalar paralelismo. O processo mestre cria processos escravos, supre requisição e espera para completar; podendo, então, destruir o escravo.

4.2.4.32 Método *Factory*

Define uma interface para a criação de um objeto, mas permite que a subclasse decida que classe instanciar; especifica a criação do esqueleto usando construtores primitivos, várias primitivas em subclasses.

4.2.4.33 Método *Template*

Define o esqueleto de um algoritmo em uma operação, cedendo alguns passos para as subclasses; permite que as subclasses redefinam certos passos de um algoritmo, sem mudar a estrutura do algoritmo; especifica o esqueleto do algoritmo usando primitivas, várias primitivas em subclasses ou pela delegação.

4.2.4.34 Módulo

É um grupo de componentes que mudam simultaneamente sem sensibilidade de interfaces não podendo ser instanciado mais de uma vez. Pode, então, mudar / repor a implementação, *hardware*, I / O, sistema operacional, recursos da memória, sem afetar os clientes, tornando a interface independente de prováveis mudanças. São suportados por interfaces Java, Modula, Ada e C.

4.2.4.35 Objeto Nulo (*Stub*)

Elimina frequentemente testes para referências nulas, pela reposição a um objeto nulo, que é uma instância de uma classe com pseudo implementações dos métodos.

4.2.4.36 *Pipeline* (*Pipes* e Filtros)

Transmite dados através de uma sequência de transformações independentes, repõem / adicionam / deletam estágios e mudam topologias; definem formatos, I / O; protocolos e adaptadores competindo com *pipes.*

4.2.4.37 *Propagator*

Propaga mudanças através da rede de objetos dependentes; estende/ escolhe a rede e adiciona novas classes de objetos da rede. É implementado para registro uniforme e notificação da interface; registro direto ou através de evento. Suas subclasses são:

- *Strict Propagator* com/sem falha.
- *Lazy Propagator.*
- *Adaptative Propagator.*

4.2.4.38 *Observer*

Define uma para muitas dependências entre objetos. Porém, quando um objeto muda de estado, todas as dependências são notificadas e atualizadas automaticamente. É um caso especial de *propagator* com somente um nível de dependência.

4.2.4.39 Protótipo

Especifica os tipos de objetos criados usando a instância prototípica e cria novos objetos pela cópia deste protótipo, mudando, assim, o protótipo.

4.2.4.40 *Proxy*

Fornece um substituto para outro objeto, a fim de controlar seu acesso; adicionar/retirar funcionalidades não planejadas transparentemente, sem afetar o objeto original ou os clientes sem subclasses; delega requisições para o original antes/depois de adicionar funcionalidades. Alguns exemplos são listados abaixo:

- *Decorator (Proxies* Cascata).
- *Buffer Proxy, Cache Proxy.*
- *Logging Proxy, Counting Proxy.*
- *Firewall* (Proteção do *Proxy).*
- Sincronização do *Proxy.*
- Acesso Remoto ao *Proxy.*

4.2.4.41 *Recoverable Distributor*

Seu objetivo é o estado replicado em sistema distribuído, mantendo a consistência e reabilitação e escolhendo a política de consistência e reconhecimento da falha. É implementado para gerenciar o estado local e global, recolher falhas locais e globais e atualizar os protocolos.

4.2.4.42 *Singleton*

Assegura que uma classe tenha somente uma instância e fornece seu ponto global de acesso, a fim de garantir a instância simples para a classe e registrar sua existência no membro estático.

4.2.4.43 *Strategy*

Define uma família de algoritmos, encapsulando cada um e fazendo as trocas e repondo/adicionando algoritmos usados. Sua implementação consiste na interface/superclasse comum para diferentes algoritmos; múltiplas implementações de algoritmos e delegação de implementação. Um exemplo é o relacionamento *View-Controller* que é um objeto que representa um algoritmo e é útil quando se quer encapsular estruturas de dados complexas.

4.2.4.44 Superclasse

Fornece a discussão uniforme de variáveis da classe pela localização da interface comum dentro da superclasse e adiciona novas subclasses e variáveis.

4.2.4.45 *Visitor*

Representa uma operação executada nos elementos da estrutura de um objeto; permite que se defina nova operação, sem mudar as classes dos elementos em cada operação, sendo apropriado quando se quer habilitar uma variedade de coisas diferentes para objetos que têm uma estrutura de classe fixa. Pode ser utilizado em operações de desligamento, existentes em variáveis para muitas classes; extensibilidade e adição de novas operações. Possui algumas variações como:

- *Default Visitor.*
- *Visitor* Externo.
- *Visitor* Acíclico.

4.2.5 *Propagation Patterns*

Propagation patterns são complementos de padrões de projeto descritos por Gamma (1994) e expressam grupos de objetos colaborando para um propósito específico; são executáveis e têm no mínimo duas aplicações: expressando *design patterns* para desenvolvimento de *software* orientado a objeto e usando padrões de projeto para guiar o projeto de *patterns propagation* e suas adaptações.

Padrões para problemas de programação orientada a objeto, podem ser descritos usando notações de *Adaptive Programming* (AP). Um excelente padrão candidato é o *Visitor* que adquire aprimoramento e formalização concisa, usando adaptações.

O *Builder* pode ser implementado usando sentenças para descrever objetos de maneira robusta, sem referência à classe específica. Já o *Prototype* pode ser implementado usando uma especificação de passagem, para definir uma operação clone. A especificação da passagem define um subgráfico que necessita ser copiado. Em geral, AP é aplicável para padrões de projeto e envolve subgráficos selecionados de grandes gráficos. A capacidade de se adaptar também é útil para qualquer padrão que envolva interações, desde que a especificação da passagem seja sucinta.

Muitos padrões de projeto para orientação a objeto são aplicáveis diretamente para AP. Por exemplo, o *Pattern Siemens Reflection* pode ser usado para implementar AP; ou o *Observer* é útil para separar modelos de *views* em AP. Em resumo, a ideia de padrão de projeto é útil para AP e o objetivo é prover abstrações que permitam expressar melhor estes padrões para OOP e AP, amenizando o efeito de muitas tarefas de projeto.

4.2.6 *Patterns* para *Adaptive Programming* (AP)

4.2.6.1 *Adaptive Dynamic Subclassing*

As ideias de AP têm sido reinventadas em diferentes domínios. E desde que a ideia tenha sido usada com sucesso, é tempo de se formular, em termos de padrões para isolar a chave das ideias e fazê-las mais fáceis. Normalmente são usados cinco tipos de padrões para descrever AP, sendo o *Inventor's Paradox* o primeiro, que usa muitos acoplamentos de forma

livre. Este padrão tem quatro *subpatterns* chamados Estrutura *Shy Traversal*, Estrutura *Shy*, Objeto e Contexto.

Estes padrões são úteis para o projeto do *software*, para projetistas que querem engrandecer suas técnicas, e metodologistas que querem adicionar adaptações a seus métodos. A Estrutura *Shy Traversal* e Contexto são aprimoramentos do padrão *Visitor*, sendo bons para descrever passagens e contextos, permitindo, assim, uma descrição elegante de objetos *visitor*. A Estrutura *Shy Traversal* distribui somente dois relacionamentos: passagens e classes de gráficos, e a Estrutura *Shy Object* usa dois relacionamentos: representações do objeto textual e classes de gráficos estendidas.

A Estrutura Contexto usa o comportamento e a modificação do comportamento.

4.2.6.2 *Adaptive Builder*

Aplicações necessitam de mecanismos para a construção de objetos complexos e independentes para as partes que criam os objetos e as ferramentas de Demeter e o Dicionário de Classes de Demeter particionam as descrições dos objetos, resolvendo, então, este problema.

Este método encontra requisições do padrão *Builder* que é o algoritmo para criação do objeto complexo, podendo ser independente das partes que fazem o objeto e da maneira que são reunidos. Assim, fazer uma aplicação robusta mecaniza a definição da classe gráfica e induz a operações semelhantes à *copying*, *displaying*, *printing*, *checking* etc.

4.2.6.3 Classe Diagrama e Classe Dicionário

É independente do mecanismo que cria o objeto, sendo aplicável para qualquer sistema que necessite de uma ou várias classes gráficas. Pode mudar o tempo, sem afetar o processo. Usando este método, não há a necessidade de se definirem as classes *Director*, *Builder* e *ConcreteBuilder*.

4.2.6.4 Estrutura *Shy Object*

Seu objetivo é fazer as descrições do objeto, independentes dos nomes das classes e mudar a estrutura das classes. Durante a evolução de uma aplicação as estruturas da classe necessitam de atualizações em seus objetos. Assim, este modelo é útil em projetos orientados a objetos para leitura e impressão de objetos, em notações definidas pelo usuário.

Um exemplo de projeto pobre que pode ser aprimorado pela Estrutura *Shy Object* é aquele que contém muitos construtores chamados para construírem objetos *composite.* É útil para o projeto de uma linguagem nova que tem controle sintático e necessita implementar o aproveitamento das classes. Estende a aplicação da estrutura da classe, porém cada objeto conhece a análise e imprime a função.

4.2.6.5 *Adaptive Interpreter*

Determinada a linguagem, define-se, então, a representação para a gramática, e será especificada pela classe recursiva dicionário; permitindo que um interpretador use a representação para as sentenças interpretadas na linguagem. Depois da criação da gramática, um interpretador é construído para interpretar sentenças da linguagem, ao invés de uma instância do problema.

Demeter usa esta gramática para criar uma parceria para a linguagem, que poderá particionar sentenças dentro das árvores do objeto e imprimir os objetos saída como sentenças na linguagem; onde cada sentença pode se dividir dentro de um único objeto.

O *Interpreter Adaptive* tem alguns benefícios como: suporte para a expansão da gramática, desde que a gramática esteja definida na classe dicionário (CD) facilitando sua modificação e para o interpretador da nova linguagem. O comportamento do interpretador é definido usando *o Propagation Pattern* (PP).

O uso da Estrutura *Shy Object* conduz a descrições mais robustas e curtas dos objetos, resultando em um teste fácil das estruturas da classe, antes do início da programação. Tem um efeito positivo no projeto da classe, pois permite testar suas estruturas para integridade na representação dos objetos. É útil na conjunção com o padrão *Prototype* e pode ser usado para criar um protótipo na estrutura *shy.*

4.2.6.6 *Adaptive Visitor*

Aprimora o padrão *Visitor*, representando uma operação que pode ser executada nos elementos da estrutura do objeto; reunião do código descrevendo a passagem com dependência mínima na estrutura da classe.

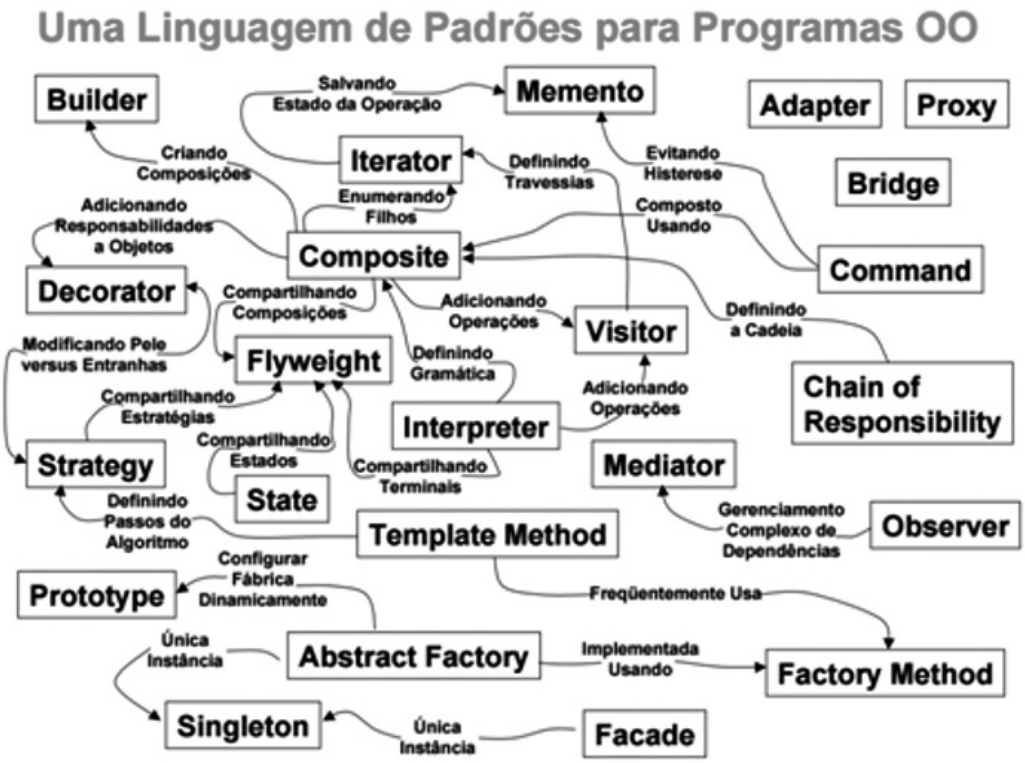
Para que uma operação seja executada na estrutura do objeto, muitos dos objetos envolvidos são casuais. Estes objetos têm uma simples passagem do comportamento e seus métodos são pequenos. *Adaptive Visitor* adiciona novas operações facilmente, usando o poder do *Iterator* através de qualquer tipo da estrutura do objeto.

Os problemas de padrões para programação orientada a objeto poderão ser descritos usando as notações de AP e um candidato é o *Visitor*, que consegue um aprimoramento significante ao usar a capacidade de se adaptar. Assim, as aplicações que queiram alterar dinamicamente a representação de um objeto, usam este modelo.

FONTE: Medeiros (2000, p. 72-81)

A figura a seguir exibe um resumo dos padrões apresentados nesta unidade!

FIGURA 10 – PADRÕES PARA PROGRAMAS OO



FONTE: Disponível em: <http://bit.ly/2wefzdR>. Acesso em: 13 fev. 2016.

Assista ao vídeo sobre o conteúdo de padrões, no seguinte endereço: <http://www.devmedia.com.br/padroes-de-projeto-iii-padroes-comportamentais-curso-extreme-programming-aula-36/34143>.

Para mais informações sobre padrões de projetos Orientados a Objetos, consulte os links:

Padrões de Design OO
<http://hillside.net/patterns/patterns.html>
<http://gee.cs.oswego.edu/dl/cpj/ifc.html>
<http://st-www.cs.uiuc.edu/cgi-bin/wikic/wikic?JavaAWT>
<http://mordor.cs.hut.fi/tik-76.278/group6/awtpat.html>

Transactions and Accounts
<http://c2.com/cgi/wiki?TransactionsAndAccounts>
<http://st-www.cs.uiuc.edu/users/johnson/Accounts.html>

Padrões para Processo de Desenvolvimento de Estrutura de Organizações
<http://www.bell-labs.com/cope/Patterns/Process/>

Padrões de Reengenharia
<http://www.iam.unibe.ch/famoos/patterns/>

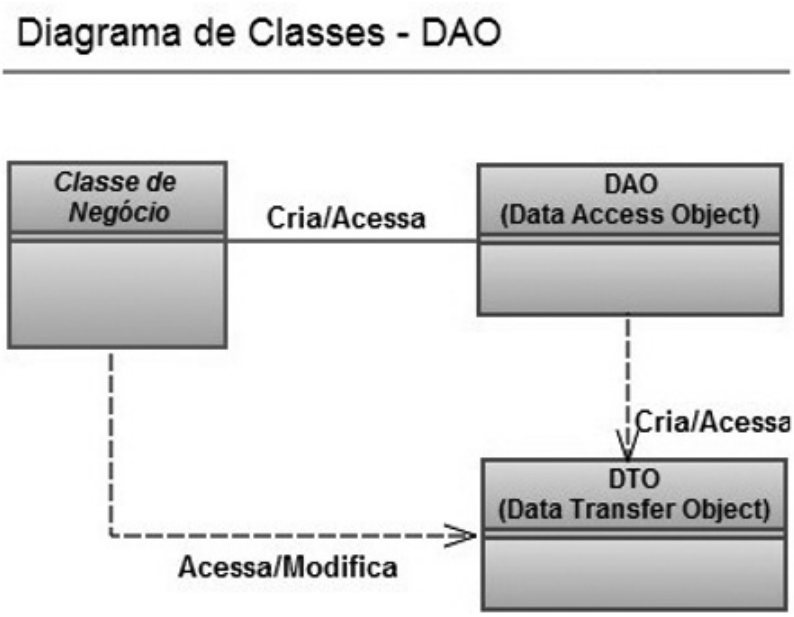
5 OUTROS PADRÕES

Outros padrões comumente usados no desenvolvimento de aplicações baseadas em orientação a objetos são os **Core J2EE Patterns** como o DAO (*Data Access Object*), BO (*Business Object*) e DTO (*Data Transfer Object*). Entre estes existem seus respectivos correspondentes em outras plataformas, como no Framework.Net respectivamente o DAL, o BLL e DTO.

5.1 DAO (*DATA ACCESS OBJECT*) OU DAL (*DATA ACCESS LAYER*)

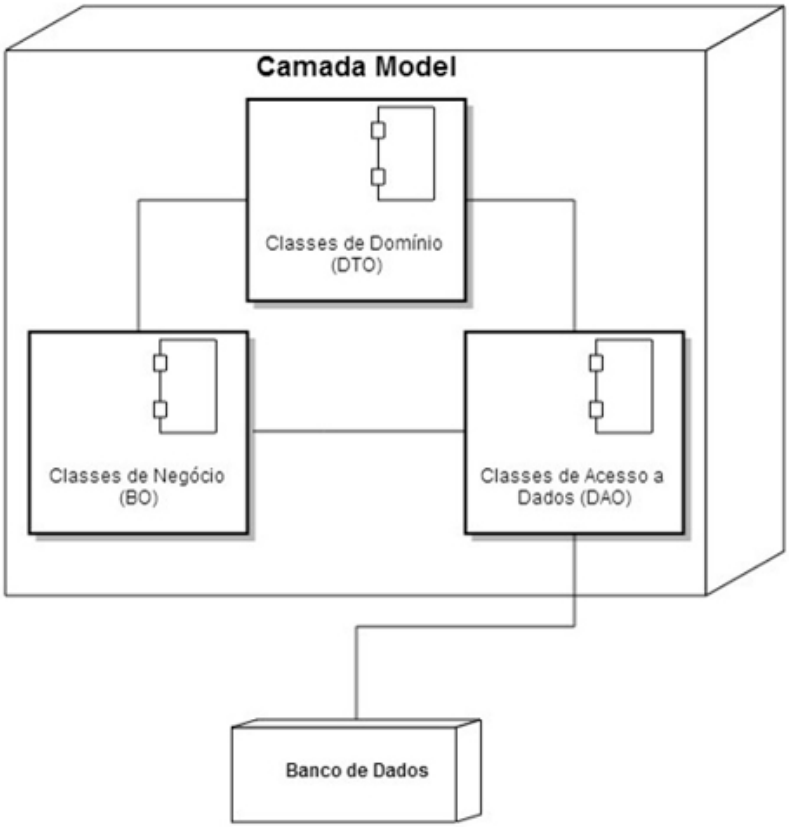
Objetivo: encapsular o acesso a dados em uma camada separada da aplicação.

FIGURA 11 – DIAGRAMA DE CLASSES - DAO



FONTE: Disponível em: <http://bit.ly/2wcsdKC>. Acesso em: 1 abr. 2015. p. 5.

FIGURA 12 – DIAGRAMA DE COMPONENTES E DE DISTRIBUIÇÃO



FONTE: Disponível em: <http://bit.ly/2wcsdKC>. Acesso em: 1 abr. 2015. p. 6.

Na **Figura 11** é exibido o diagrama de classes do padrão de projeto DAO (DAL) com indicações da **criação** ou **acesso** a objetos. A **Figura 12** exibe dois diagramas: o de componentes e o de distribuição (*deployment*). O de componentes representa, conforme o próprio nome do diagrama, os componentes da aplicação. No caso em questão, representam as **Classes de Domínio**, **Classes de Negócio** e as **Classes de Acesso a Dados**. Esta representação (camada model, componentes e banco de dados) constitui o diagrama de distribuição (*deployment*) que no caso em questão visa exibir os relacionamentos entre componentes em um nível maior de abstração.

Características:

• **Centralização do código de acesso/manipulação de dados da aplicação.** Este fato possibilita uma manutenção mais eficiente do código de persistência da aplicação, uma vez que o mesmo se encontra localizado em uma única camada. Em uma situação, por exemplo, de alteração em alguma consulta à base de dados, sabe-se que a modificação será realizada na camada que implementa o DAO, o que acarreta ganho de tempo pois evita localizar e modificar em diversas partes do código da aplicação os pontos que ocorrem a consulta em questão.

• **Separação da lógica de negócio da persistência.** O uso do DAO implica na separação das operações de banco de dados das regras de negócio, possibilitando realizar alterações nessas regras sem necessariamente alterar as ações de banco de dados relacionadas.

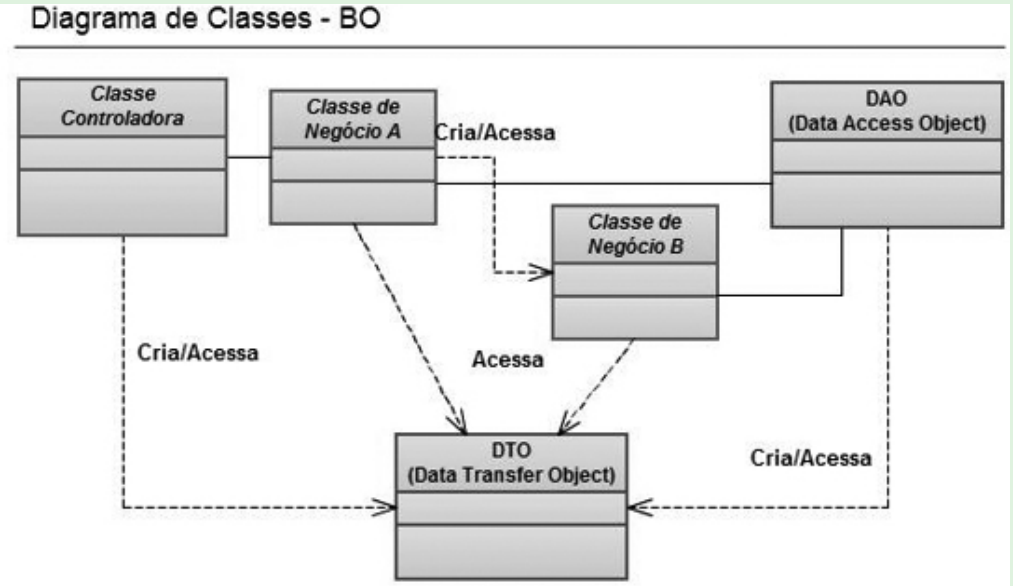
• **Tornar transparente o acesso aos dados nas aplicações.** Clientes não necessitam conhecer a forma de acesso aos dados da aplicação. Existem casos em que as operações de banco de dados são realizadas na interface do sistema, prática não recomendável, pois expõe detalhes do acesso aos dados. A criação de uma camada dedicada à persistência separada da interface ou de códigos que acessam diretamente esta interface "esconde" detalhes da manipulação de dados do usuário, aumentando assim, a segurança da aplicação.

• **Possibilitar acesso a diferentes fontes de dados de forma transparente para o usuário.** Aplicações corporativas podem necessitar acessar diferentes fontes de dados. Em determinado contexto um sistema pode acessar uma base de dados em Oracle, em outro uma base em SQL Server, em outro é necessário que acesse um arquivo xml etc. O padrão DAO em conjunto com padrões de projeto que atuam como fábricas de objetos (*Factory* e *Abstract Factory*) possibilita a implementação de acesso para diferentes mecanismos de persistência.

5.2 BO (*BUSINESS OBJECT*) OU BLL (*BUSINESS LOGIC LAYER*)

Objetivo: Centralizar as regras de negócio da aplicação.

FIGURA 13 – DIAGRAMA DE CLASSES - BO



FONTE: Disponível em: <http://bit.ly/2wcsdKC>. Acesso em: 1 abr. 2015. p. 8.

Características:

• **Centralização da lógica de negócio da aplicação**. O padrão BO visa centralizar todas as regras de negócio da aplicação. Dessa forma, evita-se que a lógica de negócio esteja dispersa ou duplicada pelo sistema, facilitando a criação e a manutenção de funcionalidades.

• **Separação da lógica de negócio do acesso a dados.** Os métodos de uma classe de negócio não contêm operações de acesso a dados (consulta, atualização, inserção e exclusão). Com este procedimento, alterações nas regras de negócio, bem como nas operações de acesso a dados podem ser realizadas de forma mais focada, reduzindo-se assim o tempo de implementação e evitando-se impactos indesejáveis no sistema.

• **Reutilização de código.** Objetos de negócio podem ser reutilizados por outras funcionalidades da aplicação, diminuindo-se assim, o tempo de codificação.

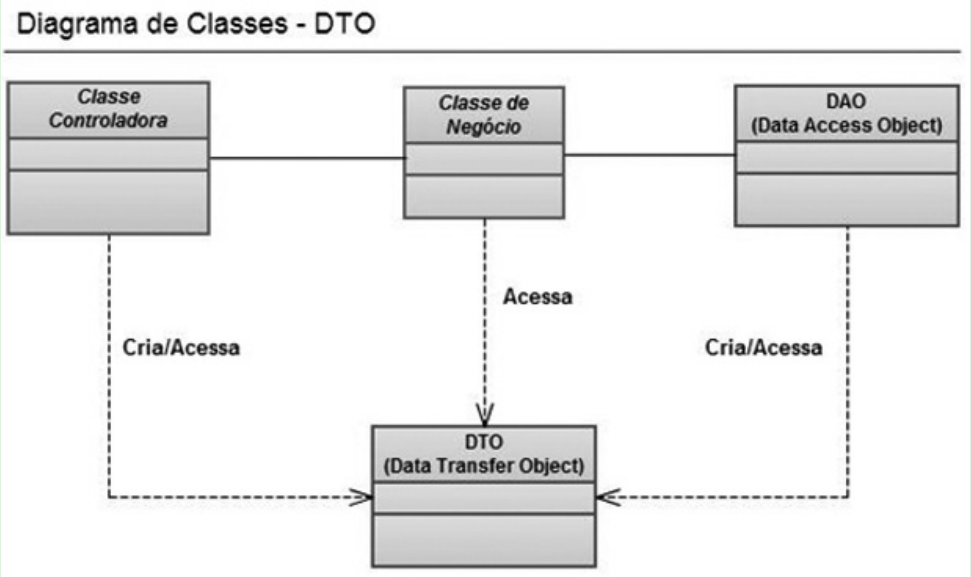
• **Possibilidade de uso em outras arquiteturas.** Uma vez que objetos de negócio encapsulam as regras de negócio de um sistema em uma única camada, esta pode ser utilizada em outras arquiteturas, como por exemplo,

em **SOA** (*Service-Oriented Architecture*). Neste caso modifica-se apenas as chamadas dos métodos de negócio para a acessarem serviços, e não mais métodos de classes de acesso a dados.

5.3 DTO (*DATA TRANSFER OBJECT*)

Objetivo: transportar diversos dados de uma única vez entre as camadas da aplicação.

FIGURA 14 – DIAGRAMA DE CLASSES - DTO



FONTE: Disponível em: <http://bit.ly/2wcsdKC>. Acesso em: 1 abr. 2015. p. 10.

Características:

• **Reduzir o tráfego na rede**. O padrão DTO configura-se como uma solução para as entidades de domínio da aplicação. Um objeto que segue este padrão concentra os atributos da entidade e seus respectivos métodos de acesso. Sendo assim, ao concentrar os dados da entidade em um único objeto diminui-se o tráfego na rede, evitando uma chamada para cada atributo de uma entidade, uma vez que seus atributos estão encapsulados.

• **Transportar dados entre camadas.** Este objeto pode ser transportado entre as camadas da aplicação pelas quais pode ser manipulado, em dado momento recebendo valores da camada de visão e

persistindo estes valores na base de dados, em outro momento coletando informações da base de dados e os transportando para a camada de visão. Ainda durante estes transportes o objeto do tipo DTO pode ter os valores de seus atributos modificados por meio das regras de negócio da aplicação.

• **Simplificação de especificação e codificação de interfaces.** Em situações em que é necessário, por exemplo, consumir um serviço ou enviar/receber objetos entre sistemas, a especificação de objetos que concentrem os parâmetros ou atributos a serem consumidos ou transportados entre aplicações, simplifica o projeto e a codificação dessas operações. O cliente, por exemplo, necessitará conhecer apenas a interface do emissor, ou seja, apenas o objeto DTO que será utilizado.

5.4 *SINGLETON*

Objetivo: criar apenas um objeto de uma determinada classe.

FIGURA 15 – DIAGRAMA DE CLASSES - SINGLETON

FONTE: Disponível em: <http://www.sefaz.ba.gov.br/downloads/prt/manuais/padroes/manual_de_padroes_desenvolvimento_web_mvc.doc>. Acesso em: 1 abr. 2015. p. 11.

Características:

• **Reutilização de objetos:** Padrão de projeto que visa propor uma solução para reutilização de objetos usados por vários outros objetos da aplicação. Por exemplo, um objeto do tipo DAO será instanciado a cada vez que for realizada uma operação com a base de dados. Numa situação destas deve-se considerar o uso de memória da máquina, pois a instanciação de objetos é um processo que aloca memória, e vários objetos do mesmo tipo realizando praticamente as mesmas tarefas ocuparão um espaço desnecessário na memória principal. Sendo assim, é interessante alocar-se espaço em memória para apenas 01 (um) objeto do mesmo tipo

e que desempenhe comumente as mesmas operações. A fim de ajudar a solucionar este problema foi criado o padrão *Singleton*. O processo de criação de objetos do tipo *Singleton* fica a cargo da aplicação.

6 MODELO MVC

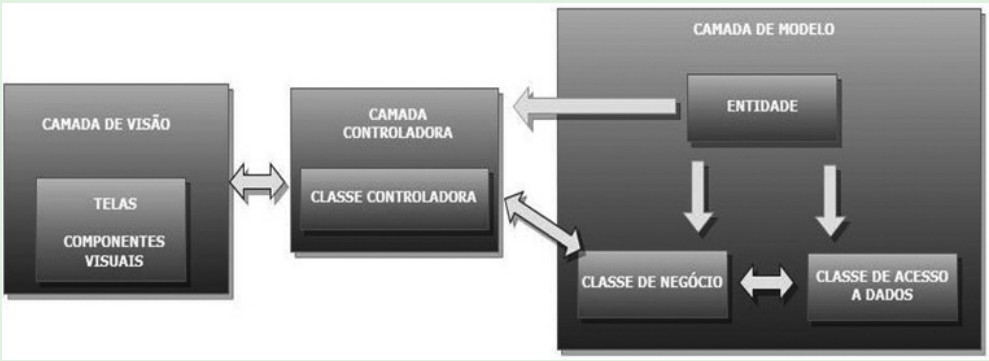
O modelo MVC é um padrão de desenvolvimento de *software* baseado em 03 camadas. Este modelo foi criado em 1979 pelo cientista da computação Trygve Mikkjel Heyerdahl Reenskaug - atualmente professor emérito da Universidade de Oslo – para a GUI (*Graphic User Interface*) de um projeto de *software*, enquanto visitava o Centro de Pesquisa da Xerox, em Palo Alto. As três camadas do MVC são:

- Modelo (Model).
- Visão (View).
- Controladora (Controller).

Esta divisão visa separar a lógica da aplicação da apresentação das informações ao usuário. Cada camada possui suas respectivas responsabilidades: a camada de Modelo concentra as classes de domínio (entidades) da aplicação, além das classes de negócio e de acesso a dados; a camada de Visão é responsável pelo *layout* da aplicação (telas em HTML, por exemplo) e seus componentes visuais; a camada Controladora tem como objetivo direcionar o fluxo de dados entre as camadas de visão e de modelo da aplicação. Dessa forma, pode-se alterar as telas ou componentes visuais do sistema sem modificar as classes responsáveis pela a lógica da aplicação (modelo e controladoras), e vice-versa. Sendo assim, diminui-se o tempo de manutenção de funcionalidades devido à alta coesão (classes com responsabilidades e objetivos bem definidos) e ao baixo acoplamento (pouca dependência de outras classes).

A figura a seguir exemplifica uma representação do modelo MVC:

FIGURA 16 – MODELO MVC



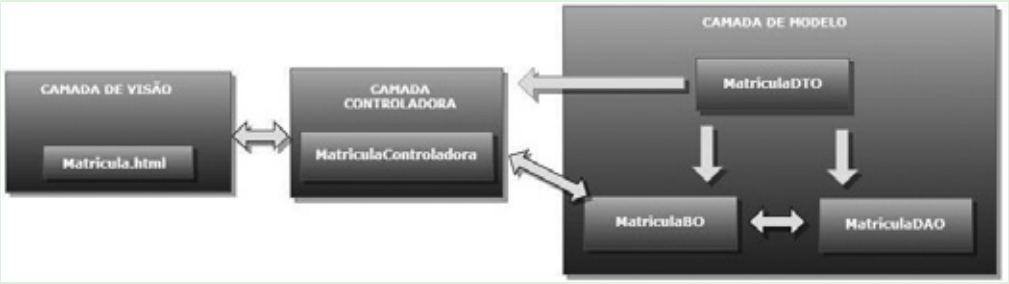
FONTE: Disponível em: <http://bit.ly/2wcsdKC>. Acesso em: 1 abr. 2015. p. 13.

Uma desvantagem do modelo MVC é que praticamente a cada nova funcionalidade será necessário criar as respectivas camadas de visão, controladora e modelo. Mas após a conclusão dessas atividades há uma redução significativa na manutenção, conforme descrito anteriormente.

Como exemplo, supõe-se que em uma faculdade exista um sistema, composto por módulos, que controla suas atividades. O módulo de matrículas, de acordo com o modelo MVC, deve ser constituído de (vide figura a seguir):

- Tela de Matrícula (Matricula.html)
- Classe controladora de Matrícula (MatriculaControladora)
- Entidade de Matrícula (Matricula)
- Regras de negócio de Matrícula (MatriculaRN (Regras de Negócio))
- Persistência de Matrícula (MatriculaPersistente)

FIGURA 17 – MÓDULO DE MATRÍCULAS SEGUINDO O MODELO MVC



FONTE: Disponível em: <http://www.sefaz.ba.gov.br/downloads/prt/manuais/padroes/manual_de_padroes_desenvolvimento_web_mvc.doc>. Acesso em: 1 abr. 2015. p. 14.

Pode-se combinar o MVC com padrões de projeto, como por exemplo, os mais comumente usados: DAO (*Data Access Object*), DTO (*Data Transfer Object*), BO (*Business Object*) e *Singleton*. Fazendo ainda uso do exemplo do módulo de matrícula, aplica-se o padrão de projeto DTO na entidade Matrícula, que passa a ser denominada MatriculaDTO. A classe MatriculaRN recebe o padrão BO, tornando -se MatriculaBO e os padrões DAO e *Singleton* são aplicados na classe MatriculaPersistente, sendo denominada MatriculaDAO.

7 PADRÕES DE CODIFICAÇÃO

Atividades relacionadas ao processo de desenvolvimento de *software* podem ser bastante complexas dependendo dos objetivos e contextos para os quais o sistema é elaborado. Padrões de projeto foram concebidos com a finalidade de diminuir essa complexidade tanto nas atividades de modelagem quanto na codificação de aplicações. Além do uso de padrões de projeto, tarefas padronizadas de codificação possibilitam uma melhor compreensão das funcionalidades a serem ou já implementadas, pois todos os membros da equipe seguem a mesma forma de programação, reduzindo-se assim o tempo de criação e principalmente o de manutenção do sistema. A combinação de padrões de projeto e tarefas padronizadas de codificação possibilitando a construção de um *software* mais consistente, com reutilização de código e componentes e de manutenção facilitada. Seguem técnicas para padronização da codificação:

Nomenclatura:

Evitar:

for (int **i** = //Código a ser executado).

Neste caso utilizar, por exemplo: for (int **contadorCamposObrigatorios** =//Código a ser executado)

• Evitar o uso de variáveis globais, com exceção de objetos do padrão *Singleton*, desde que devidamente planejados e implementados (vide tópico *Singleton*). Atributos que possuam valores fixos ou poucas vezes alterados durante a execução do sistema – e que geralmente são fortes candidatos a variáveis globais – devem ser convertidos em constantes. Exemplo:

FIGURA 18 – EXEMPLO DE PADRÃO DE CODIFICAÇÃO

```
if (documentacao.CodVersaoPrincipal.Equals("0"))
{
    |
    //Código a ser executado
}
```

FONTE: Disponível em: <http://www.sefaz.ba.gov.br/downloads/prt/manuais/padroes/manual_de_padroes_desenvolvimento_web_mvc.doc>. Acesso em: 1 abr. 2015. p. 17.

FIGURA 19 – EXEMPLO DE PADRONIZAÇÃO DE CODIFICAÇÃO PARA AUTENTICAÇÃO

```
string CODIGO_AUTENTICACAO_SUCESSO = "0";

if (documentacao.CodVersaoPrincipal.Equals(CODIGO_AUTENTICACAO_SUCESSO))
{

    //Código a ser executado
}
```

FONTE: Disponível em: <http://www.sefaz.ba.gov.br/downloads/prt/manuais/padroes/manual_de_padroes_desenvolvimento_web_mvc.doc>. Acesso em: 1 abr. 2015. p. 18.

Na situação demonstrada acima, a primeira instrução if testa se o código retornado pelo objeto retornoDTO é igual a zero (vide **Figura 18**). Mas o que significa para a aplicação esse código zero? Esta forma de programação pode dificultar a manutenção da funcionalidade relacionada. Verifique que o nome atribuído à variável facilita a compreensão da escolha do nome atribuído à mesma (vide **Figura 19**). As constantes relacionadas a uma funcionalidade da aplicação podem estar localizadas em uma interface iniciada com a letra "I" maiúscula. Exemplo:

- Classe: **ControleAcesso**
- Interface com constantes da classe ControleAcesso: **IControleAcesso**

Constantes ou parâmetros utilizados em vários pontos da aplicação, bem como as mensagens a serem exibidas para o usuário, podem estar armazenados em uma interface ou em arquivo de configuração, como do tipo XML. Essa técnica de centralização de constantes/parâmetros em um único arquivo – interface ou um arquivo de configuração, propriedades – possibilita uma manutenção mais rápida em caso de alterações, pois evita-se localizar em toda a aplicação os pontos que devem ser alterados, além de evitar o risco de modificar-se seus valores em uma funcionalidade e em outra não. Além disso, a **programação orientada a interfaces** possibilita que a construção do sistema siga definições pré-estabelecidas como assinaturas de métodos genéricos (considerando ainda seus tipos de parâmetro e retorno). O uso de interfaces estabelece uma forma de "contrato" que deve servir como norteador da implementação da aplicação, e dessa forma, procurar evitar que aspectos genéricos sejam construídos ao gosto do desenvolvedor.

Evitar usar mais de uma vez a instrução de retorno, geralmente denominada ***return*.** Esta prática facilita a compreensão da operação que está sendo realizada pelo método, e dessa forma, agiliza os testes deste método bem como sua depuração (***debug***).

FIGURA 20 - DUPLICIDADE DE RETORNO NA CODIFICAÇÃO

```
public string retornarCodVersaoRevisao(DocumentacaoDTO documentacao)
{

    if(documentacao.CodVersaoRevisao.Equals("0")){

        string retorno = null;

        return retorno;
    }

    if(documentacao.CodVersaoRevisao.Equals("1")){

        string retorno = null;

        return retorno;
    }

    if (documentacao.CodVersaoRevisao.Equals("2"))
    {

        string retorno = null;

        return retorno;
    }
```

FONTE: Disponível em: <http://www.sefaz.ba.gov.br/downloads/prt/manuais/padroes/manual_de_padroes_desenvolvimento_web_mvc.doc>. Acesso em: 1 abr. 2015. p. 19.

Na situação apresentada na **figura acima** será necessário depurar os diversos trechos do código a fim de verificar qual valor será retornado ao final da execução do método. Um outro exemplo para o mesmo método:

FIGURA 21 – OUTRO EXEMPLO PARA CODIFICAR O RETORNO

```
public string retornarCodVersaoRevisao(DocumentacaoDTO documentacao)
{

    string retorno = null;

    if(documentacao.CodVersaoRevisao.Equals("0")){

        retorno = "0";
    }

    if(documentacao.CodVersaoRevisao.Equals("1")){

        retorno = "1";
    }

    if (documentacao.CodVersaoRevisao.Equals("2"))
    {

        retorno = "2";
    }

    return retorno;
}
```

FONTE: Disponível em: <http://www.sefaz.ba.gov.br/downloads/prt/manuais/padroes/manual_de_padroes_desenvolvimento_web_mvc.doc>. Acesso em: 1 abr. 2015. p. 20.

Para o segundo exemplo (**Figura 21**) observa-se o aumento de legibilidade do código pois é usada a instrução ***return*** ao final do método. Para tanto, é criada uma variável **retorno** que recebe o valor do tipo esperado no retorno do método e que é populada no decorrer da execução do mesmo. Este procedimento também facilita a depuração pois pode-se ir direto à instrução ***return*** com o objetivo de conhecer o valor do retorno do método, armazenado na variável **retorno.**

Métodos:

- Métodos devem possuir apenas um único objetivo e dessa forma, todo seu código deve estar focado nesse objetivo. Por exemplo, um método **getDocumentacao** na camada controladora deve possuir, em termos de código para retornar uma lista de objetos **Documentacao**, apenas a chamada para o respectivo método na camada de negócio. Na camada de negócio, o método **obterTodos** deve conter as regras de negócio pertinentes à operação de retorno dos objetos em questão e/ou a chamada para o respectivo método na camada de acesso a dados. Obrigatoriamente, todas as validações da aplicação (como campo

requerido, tamanho de campo etc.) têm que ser realizadas na classe de negócio e na interface. De acordo com esta obrigatoriedade, no caso de aplicações *web*, as validações devem ser realizadas tanto nas classes de negócio (**lado servidor**) como nas páginas html (**lado cliente**). Na camada de acesso a dados, o método responsável pelo retorno da lista de objetos deve possuir apenas código relacionado ao retorno da lista citada, sem possuir lógica de negócio (veja **Figuras 22**, **23**, **24**):

FIGURA 22 - EXEMPLO DE MÉTODO - 1

```
namespace ExemploASLibraryLayers.Controllers
{
    class DocumentacaoController
    {

        [DataObjectMethodAttribute(DataObjectMethodType.Select, true)]
        public ListaDocumentacaoDTO getDocumentacao()
        {
            return new DocumentacaoBLL().obterTodos();
        }

    }
}
```

FONTE: Disponível em: <http://bit.ly/2wcsdKC>. Acesso em: 1 abr. 2015. p. 21.

FIGURA 23 – EXEMPLO DE MÉTODO - 2

```
public class DocumentacaoBLL
{

    public ListaDocumentacaoDTO obterTodos()
    {
        try
        {

            return new DocumentacaoDAL().obterTodos();

        }
        catch (Exception ex)
        {
            throw new ASLibrary.Util.Mensagens(ex);
        }

    }
```

FONTE: Disponível em: <http://bit.ly/2wcsdKC>. Acesso em: 1 abr. 2015. p. 21.

FIGURA 24 – EXEMPLO DE MÉTODO - 3

```
public class DocumentacaoDAL : IDocumentacaoDAL
{

    public ListaDocumentacaoDTO obterTodos()
    {
        ArrayList array = new ArrayList();
        ListaDocumentacaoDTO Lista = new ListaDocumentacaoDTO();

        try
        {
            //Select para retornar uma coleção contendo todos os registros do banco de dados do DTO em questão
            SqlSelect(null);

            array = DAO.RetornaArrayList();

            for (int contadorAtributosDocumentacao = 0; contadorAtributosDocumentacao < array.Count; contadorAtributosDocumentacao++)
                Lista.Add
                (
                    new DocumentacaoDTO()
                    {

                        SeqControleDocumentacao = DAO.GetInt(array, contadorAtributosDocumentacao, 0),
                        CodVersaoPrincipal = DAO.GetStr(array, contadorAtributosDocumentacao, 1),
                        CodVersaoSecundaria = DAO.GetStr(array, contadorAtributosDocumentacao, 2),
                        CodVersaoRevisao = DAO.GetStr(array, contadorAtributosDocumentacao, 3),
                    }
                );

            return Lista;
        }
        catch (Exception ex)
        {
            //Lançar exceção referente a erro de banco de dados
            throw new ASLibrary.Util.Mensagens(ex);
        }
```

FONTE: Disponível em: <http://www.sefaz.ba.gov.br/downloads/prt/manuais/padroes/manual_de_padroes_desenvolvimento_web_mvc.doc>. Acesso em: 1 abr. 2015. p. 22.

- A codificação de métodos deve ser clara o suficiente para compreensão adequada dos seus objetivos. Além das recomendações citadas neste documento, observar se a complexidade de determinada regra dificultaria seu entendimento e posterior manutenção; neste caso esta regra deveria ser precedida de comentários ou removida para outro método (métodos devem ser objetivos; esta objetividade melhora a compreensão das tarefas executadas pelos mesmos). Observar também se entre métodos existem trechos repetidos de código ou código que pode ser agrupado de acordo com sua semântica; caso exista, verificar a possibilidade de criar um método com o trecho repetido ou com código que pode ser agrupado, promovendo assim, a reutilização de código bem como melhorando a compreensão do mesmo. A criação de métodos a partir de agrupamento de código que possui um objetivo em comum corresponde à técnica de refatoração *Extract Method (110)* (MARTIN, 1999).

Classes:

- Nomes de classes - desde que não sejam nativas da plataforma - também devem ser formadas com palavras em português. Seus nomes não podem conter verbos uma vez que não executam operações, operações (métodos) são invocadas a partir de objetos - instâncias - que representam essas classes. Uma classe que represente uma determinada entidade de uma fonte de dados (por exemplo, uma tabela em uma base de dados relacional) pode possuir o mesmo nome ou o mais próximo possível. Exemplos:

Entidade (tabela) no banco de dados relacional: **controle_documentacao**
Classe que representa a entidade: **ControleDocumentacao**

Entidade (tabela) no banco de dados relacional**: dbo_projeto**
Classe que representa a entidade: **Projeto**

Classes que representam exceções devem iniciar com a palavra "Excecao". Exemplos: **ExcecaoResponsavelImcompativelSituacao**,**ExcecaoCamposObrigatorios.**

- As classes controladoras não devem possuir instruções condicionais - **instruções if** - com a exceção do caso de ser necessário realizar testes em objetos próprios da classe controladora, como objetos de sessão ou requisição. Nos demais casos, como verificação de nulidade de campos (atributos) originados das telas da aplicação, as respectivas instruções condicionais devem estar na camada de negócio. Exemplo:

FIGURA 25 – CLASSES CONTROLADORAS

```
class DocumentacaoController
{

    public void inserirDocumentacao(DocumentacaoDTO documentacao)
    {

        new DocumentacaoBLL().inserir(documentacao);

    }
```

FONTE: Disponível em: <http://www.sefaz.ba.gov.br/downloads/prt/manuais/padroes/manual_de_padroes_desenvolvimento_web_mvc.doc>. Acesso em: 1 abr. 2015. p. 24.

Observa-se que no código da **figura acima** não há testes de nulidade para os atributos do objeto a ser inserido (DocumentacaoDTO). A validação de nulidade localiza-se na classe de negócio **DocumentacaoBLL** sendo realizada pelo método privado **validarCamposObrigatoriosDocumentacaoInclusao,** o qual é chamado no método **inserir**, conforme pode ser visto na **figura a seguir**:

FIGURA 26 – CLASSE DE NEGÓCIO

```
public class DocumentacaoBLL
{

    public void inserir(DocumentacaoDTO documentacao)
    {

        try
        {

            validarCamposObrigatoriosDocumentacaoInclusao(documentacao);

            new DocumentacaoDAL().inserir(documentacao);

        }
        catch (ExcecaoCamposObrigatoriosDocumentacao exCamposObrig)
        {
            throw new ASLibrary.Util.Mensagens(exCamposObrig);
        }

    }
```

FONTE: Disponível em: <http://www.sefaz.ba.gov.br/downloads/prt/manuais/padroes/manual_de_padroes_desenvolvimento_web_mvc.doc>. Acesso em: 1 abr. 2015. p. 24.

- As classes da camada de acesso a dados devem somente possuir código referente a operações relacionadas com a fonte de dados (consulta, atualização, inserção e exclusão). As regras de negócio da aplicação devem estar localizadas nas classes da camada de negócio, as quais fazem chamadas aos métodos da camada de acesso a dados.

Exceções:

- Validações - de nulidade, regras de negócio ou erros - devem prioritariamente ser tratadas com o uso de **Exceções.** A Exceção é uma estrutura presente em linguagens orientadas a objeto que visa uma estruturação mais organizada do tratamento de erros de uma aplicação. O uso correto do tratamento de exceções evita que a execução de um sistema seja interrompida de forma abrupta para o usuário, devido a uma operação que executou uma divisão por zero, por exemplo. Uma situação em que um sistema é baseado em MVC, as regras de negócio ou nulidade podem ser validadas por meio de exceções. Segue o exemplo na figura a seguir:

FIGURA 27 – TRATAMENTO DE EXCEÇÕES

```
namespace ExemploASLibraryLayers.Models.Excecao
{
    [Serializable]
    class ExcecaoCamposObrigatoriosDocumentacao : SystemException
    {

       public ExcecaoCamposObrigatoriosDocumentacao() : base() { }
       public ExcecaoCamposObrigatoriosDocumentacao(string message) : base(message) { }

    }
}
```

FONTE: Disponível em: <http://www.sefaz.ba.gov.br/downloads/prt/manuais/padroes/manual_de_padroes_desenvolvimento_web_mvc.doc>. Acesso em: 1 abr. 2015. p. 25.

FIGURA 28 – VALIDAÇÃO DE OBRIGATORIEDADE

```
private void validarCamposObrigatoriosDocumentacaoInclusao(DocumentacaoDTO documentacao){

    if (String.IsNullOrEmpty(documentacao.CodVersaoPrincipal))
    {

        throw new ExcecaoCamposObrigatoriosDocumentacao("mensagem");

    }

}
```

FONTE: Disponível em: <http://www.sefaz.ba.gov.br/downloads/prt/manuais/padroes/manual_de_padroes_desenvolvimento_web_mvc.doc>. Acesso em: 1 abr. 2015. p. 25.

A figura acima exibe um método cujo objetivo é validar o preenchimento de campos obrigatórios, o qual está localizado na camada de negócio. De acordo com o exemplo, caso o teste de nulidade ou vazio do atributo **CodVersaoPrincipal** do objeto **DocumentacaoDTO** esteja nulo ou vazio, é lançada a exceção **ExcecaoCamposObrigatoriosDocumentacao** com uma mensagem associada.

FONTE: BAHIA (2013, p. 5-25)

LEITURA COMPLEMENTAR

O IMPACTO DOS PADRÕES DE PROJETO EM VINTE ANOS

Boas receitas prontas de projetos de *softwares* são difíceis de encontrar. Cada cenário tem sua particularidade. Princípios gerais de *design* podem ser um guia, mas a realidade nos deixa em conflito com os nossos objetivos: flexibilidade e facilidade de manutenção contra tamanho e complexidade. Da mesma forma, as bibliotecas de código podem ajudar a não ter que reinventar a roda, mas na visão dos desenvolvedores menos habilidosos, o reuso de componentes prontos ainda é ficção.

Padrões de Projeto ajudaram a amenizar estes problemas, documentando soluções bem elaboradas para problemas que ocorrem repetidamente em um contexto. Em vez de apresentar um trecho de código ("*copy and paste*"), os padrões discutem as forças que impactam no *design* da solução. São exemplos de forças o desempenho e a segurança em aplicações *web*: algoritmos de criptografia e decriptografia melhoram a segurança, mas aumentam a sobrecarga no processamento. Ward Cunningham uma vez descreveu que os padrões são como irmãos mais velhos, eles te ensinam como fazer a coisa certa.

Embora os padrões tenham se tornado populares, o seu impacto como uma técnica de *design* é mais difícil de quantificar do que o impacto de um produto de *software* específico (que são as partes analisadas por este artigo). Estas partes destacam a amplitude dos padrões disponíveis 20 anos após as primeiras conferências sobre *Patterns Languages of Programming* (PLoP) e qual o impacto que alguns padrões tiveram sobre o *software* de código aberto.

Como tudo começou

O arquiteto de construções e filósofo Cristopher Alexander inspirou Kent Beck e Ward Cunningham a escreverem os primeiros padrões em 1987 para desenhar telas Smalltalk. Em 1993, Beck and Grady Booch patrocinaram um encontro no Colorado que provocou a formação da organização sem fins lucrativos Hillside Group para promover uma série de conferências sobre *Patterns Languages of Programming* (PLoP), que está comemorando 20 anos de sucesso. Estas conferências seguem um estilo altamente colaborativo com foco em "evangelização" através de oficinas, exposições de estudos de caso e *feedbacks*. Muitos padrões e livros de sucesso sugiram depois deste processo.

Em 1994, Erich Gamma e seus colegas levaram o conceito de padrões de projeto a um público amplo; no modelo original, o livro já vendeu mais de 500.000 cópias em 13 idiomas. Dois anos depois, Frank Buschmann e seu colega produziram o primeiro volume da série *Pattern-Oriented Software Architecture* e logo em seguida Martin Flowler lançou *Analysis Patterns* (recursos para leitura estão disponíveis em outros lugares). O aparente sucesso dos modelos dos

padrões de projeto fazia alguns autores adicionarem de forma gratuita aos seus títulos de publicação a palavra "*patterns*" - isso era o preço do sucesso. Em 2013, nas estatísticas das pesquisas por livros no *site* da Amazon, no setor de tecnologia, a palavra "*pattern*" foi pesquisada 5.500 vezes, por usuários distintos (incluindo um menor número de falsos positivos na detecção de padrões visuais).

A popularidade dos padrões veio cedo, e as pessoas perceberam que os padrões não vieram para substituir as habilidades dos projetistas e nem para resolver todos os problemas. Ainda assim os padrões bem descritos oferecem uma valiosa referência com base em experiências reais. Porque aprender fazendo (aprendendo com os erros), muitas vezes não é uma boa opção para os projetos do mundo real, os padrões podem fornecer uma maneira de aprender com a experiência dos outros (e erros, podem fazer bons *antipatterns*).

Nenhum sinal de enfraquecimento dos padrões

A ampla diversidade de Padrões dificulta a determinação do número exato de Padrões de Projeto documentados. Linda Rising listou mais de 1000 padrões na publicação The Pattern Almanac 2000. As conferências de PLoP promovidas pelo Grupo de Hillside (<www.hillside.net>), aceitaram mais de 1.500 padrões documentados. A taxa de submissão de novos padrões a estas conferências tem sido constante, em cerca de 100 documentos por ano. Existe uma estimativa de quatro padrões por documento, mais todos os livros que abrangem desenvolvimento de aplicativos móveis, sistemas adaptativos, arquiteturas sustentáveis, padrões de domínios específicos, meta arquiteturas, *workflow*, sistemas de tolerância à falha e segurança.

Muitas pessoas aceitam a definição de um padrão como uma solução comprovada para um problema em um contexto. Em *The Timeless Way of Building*, Christopher Alexander esclarece que "o Padrão é, em resumo, ao mesmo tempo uma coisa e a regra que define como esta coisa é criada". Padrões representam uma solução reutilizável, fornecem informações sobre a sua utilidade, vantagens, desvantagens e documentam as melhores práticas comprovadas.

Por exemplo, muitas arquiteturas de integração incluem o padrão *Broker*, que atua como um intermediário entre clientes e servidores e lida com o roteamento de mensagens, incluindo serialização e desserialização de conteúdo da mensagem. A estrutura de comunicação *Web* implementa este padrão; motores de *workflow* como o YAWL (*Yet Another Workflow Language*) também têm uma implementação rica deste padrão.

Muitos padrões são parte de uma biblioteca de padrões; exemplos incluem <https://developer.yahoo.com/ypatterns≥ e <www.securitypatterns.org≥. Muitas empresas, incluindo Amazon, Google, IBM, Lucent, Microsoft, Oracle e Siemens, têm coleções de padrões semelhantes, alguns dos quais estão disponíveis em livros e em *sites*. Um exemplo de uma coleção é o catálogo de padrões para ebusiness da IBM. Entre muitos projetos recorrentes, podem ser

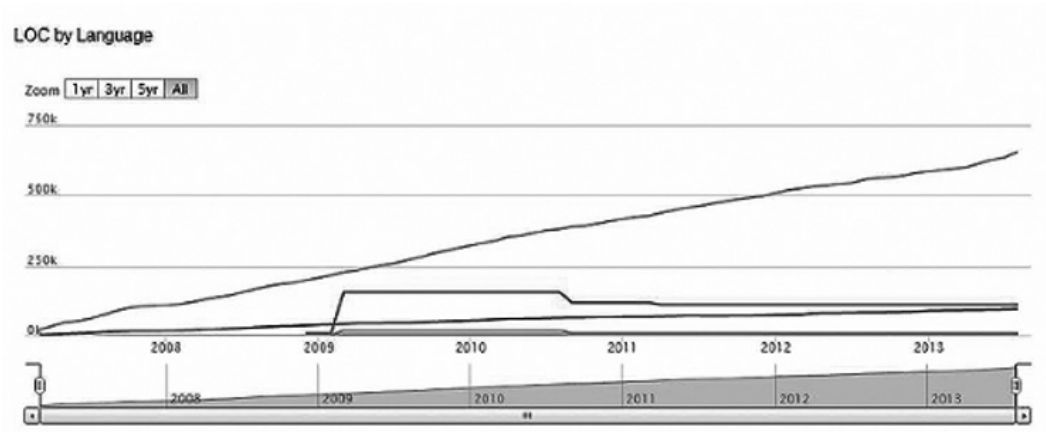
destacadas as implementações do *Enterprise Service Bus* no contexto dos produtos IBM WebSphere. Inter-relacionando e conectando um conjunto de padrões, pode ser criado um padrão de linguagem que se torna um gerador de soluções para um domínio específico durante o processo de desenvolvimento. Existe até um padrão para a escrita de padrões de projeto.

Padrões de integração corporativa

O sucesso dos padrões nas disciplinas de arquitetura de *software* e *design* tem motivado a tentativa de integrá-los em ferramentas de programação para aumentar a produtividade e ampliar a padronização em práticas de *design*. Infelizmente, a maioria das tentativas tropeçaram, porque os padrões são culturalmente associados ao contexto de documentação para a passagem de conhecimento entre as pessoas, e não como uma ferramenta de construção. Ainda assim, alguns padrões foram construídos sob influência direta da construção.

Por volta de 2003, o termo *Enterprise Service Bus* (ESB) ganhou força para descrever a plataforma de integração para arquiteturas orientadas a serviço. Rotas ESBs, filtros e transformações de mensagens XML entre os serviços, representam a evolução dos produtos de integração de aplicativos corporativos tradicionais, que implementam o padrão *Broker*. Ironicamente, embora estes produtos tenham como objetivo integrar todo o conjunto de aplicações empresariais que não se comunicavam, não existia um vocabulário comum para descrever as soluções já disponíveis (ausência dos padrões).

FIGURA 1 – CRESCIMENTO DO CÓDIGO DO APACHE CAMEL DURANTE O TEMPO



Language Breakdown

Language	Code Lines	Comment Lines	Comment Ratio	Blank Lines	Total Lines	Total Percentage
Java	654,464	306,908	31.9%	165,370	1,126,742	77.5%
JavaScript	113,619	14,668	11.4%	6,476	134,763	9.3%

O crescimento linear do código Java mostra que o engajamento da comunidade de desenvolvedores (*commiters*) é estável e sustentável. A quantidade de JavaScript saltou em 2009, porém mais tarde foi reduzida. Provavelmente com a disponibilidade de bibliotecas e *frameworks*.

Visando superar esta lacuna, os desenvolvedores de ESB´s de código aberto, disponibilizaram os *Enterprise Integration Patterns* (EIPs) que fornecem um vocabulário coerente com 65 padrões, que vão desde estilos de integração, mensagem, roteamento e transformação. Estes padrões descrevem uma parcela significativa das soluções ESB. Com a ausência de padrões na indústria de ESB´s, os projetos de código aberto assumiram a EIP como padrões de fato.

ESB´s de código aberto

Desde o surgimento em 2005, a maioria dos ESB´s de código aberto tem incorporado modelos de programação, modelos de soluções e outros modelos baseados no EIP. Os exemplos mais comuns são: Mule, Apache Camel, WSO2 ESB, Spring Integration e OpenESB.

A natureza dos projetos de código aberto faz a análise do tamanho do código ser relativamente fácil. No entanto, a análise do volume de *downloads* realizados no código é relativamente difícil, porque não existe um registro formal de quem baixa e o número de *downloads* é mascarado por espelhamento ou *caching* automáticos. O Apache Camel tem 890 mil linhas de código. Criado por 62 *commiters*, tem mais de 18.000 *commits* individuais ao longo de seis anos. É surpreendente o crescimento linear do número de linhas de código (Java), o que sugere o envolvimento estável e consistente de um conjunto de *commiters*. Adaptações comerciais ao código do Apache Camel, que desenvolveram ferramentas administrativas em tempo de execução, por exemplo pela Red Hat ou Talend, aumentaram significativamente as linhas de código.

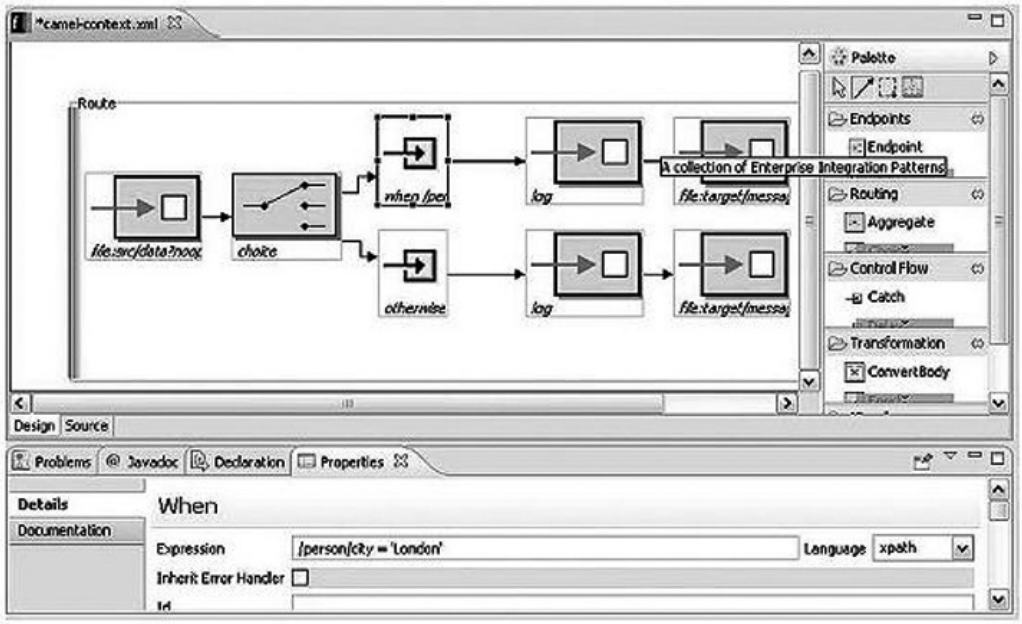
O Maven Central tem uma média de 25.000 *downloads* de *snapshots* por mês e um pico de mais de 30 mil em julho de 2013. Este volume é maior que do YAWL, que relatou cerca de 1000 *downloads* por mês em 2010. O Mule relata 3,6 milhões de *downloads* em sua página, mas não informa se todos os *downloads* são únicos.

A participação da comunidade fornece outra métrica do sucesso do código aberto. O tráfico de mensagens do Apache Camel cresceu rapidamente após o seu lançamento em 2007 e mantém-se estável com cerca de 2.500 mensagens por mês. Isso indica uma comunidade saudável que colabora para resolver problemas e impulsiona a evolução do produto. Para efeito de comparação, a página da comunidade do Mule conta com mais de 150.000 membros, e seu fórum conta com um total de 26.600 *posts*.

Padrões como uma ferramenta de *design*

Após ser integrada nestes produtos de código aberto, a EIP ficou muito popular. Alguns projetos de ESB utilizam a EIP como uma linguagem visual para o *design* de soluções. Por exemplo, um desenvolvedor pode acessar uma EIP através de um ícone no IDE Red Hat Fuse (ambiente de desenvolvimento integrado) ou Mule Studio. Ao contrário de antes, na qual houve algumas tentativas de "programação visual", o estilo arquitetônico das soluções de mensagens assíncronas torna a composição visual dos padrões mais natural. A figura a seguir mostra um Camel Rote apontando uma mensagem de entrada para dois possíveis endpoints de saída via Message Router. Desenvolvedores ESB agora podem pensar desenhar, comunicar e implementar suas soluções utilizando EIP, mesmo que estejam utilizando diferentes plataformas de integração.

FIGURA 2 – CRIAÇÃO DA SOLUÇÃO DE MENSAGENS USANDO A LINGUAGEM VISUAL DE ENTERPRISE INTEGRATION PATTERNS (EIPS)7 DENTRO DO IDE REDHAT FUSE (PLATAFORMA DE DESENVOLVIMENTO INTEGRADO)



As mensagens que chegam a partir de um *endpoint* baseado em arquivo são encaminhadas por um roteador baseado em conteúdo para um dos dois potenciais *endpoints* de saída, que roteia com base na cidade especificada dentro do conteúdo da mensagem.

FIGURA 3 – AS CARTAS DE JOGO COM BASE EM EIPS

A linguagem padrão visual permite um uso interativo dos padrões. Cada carta apresenta o ícone de um padrão juntamente com a descrição e o nome da solução.

O baralho de cartas apresentado na inauguração da conferência CamelOne é provavelmente a adaptação mais criativa do evento. Cada carta apresenta um padrão, juntamente com a descrição da solução. É gratificante ver os padrões de *design*, que foram criados para melhorar a comunicação e colaboração, encontrando seu caminho nas mãos de arquitetos e engenheiros de uma maneira acessível e útil.

As estatísticas apresentadas indicam que os Padrões de Projeto tiveram um grande impacto sobre a comunidade de *design* de *software* ao longo dos últimos 20 anos. No entanto, muitas questões em torno dos padrões ainda permanecem abertas. Por exemplo, bons padrões não são fáceis de encontrar, o que estimula a organizar e catalogar a grande quantidade de padrões existentes. Também encontramos diversas ferramentas de criação de Padrões de Projeto, que talvez utilizem tecnologia Wiki.

Finalmente, ferramentas de *design* baseadas em padrões prometem ser mais atraentes para os engenheiros de *software* e não serão mais meras ferramentas de desenho de componentes e conectores. Será que a comunidade de padrões está perdendo força? Não pensamos assim: os Padrões de Projeto existentes continuarão a ser implementados, assim como as EIP´s. E ainda existem contextos para os quais os padrões ainda não foram capturados. Por exemplo, alguns tipos de interações entre aplicações e entre as pessoas (por exemplo, através das redes sociais), poderiam ser catalogados como uma forma padrão. O futuro dos Padrões de Projeto é brilhante. Te convidamos a ajudar a moldá-lo, promovendo o desenvolvimento de uma ferramenta ou escrevendo e compartilhando o seu conhecimento em uma forma padrão.

FONTE: Disponível em: <http://www.infoq.com/br/articles/twenty-years-of-patterns-impact>. Acesso em: 29 fev. 2016.

RESUMO DO TÓPICO 3

Neste tópico você viu que:

- O conceito de padrões.
- Que a utilização de padrões na área de desenvolvimento de *software* vem crescendo bastante e em diversas fases do processo de desenvolvimento.
- Os conceitos de padrões e conhecemos seus principais elementos essenciais: nome, problema, solução e consequência.
- Aprendemos como um padrão é importante para a distribuição de conhecimento.
- Aprendemos sobre modelos, o que são modelos conceituais e como são importantes na realização de análise de negócios.
- A origem dos padrões de análise e sua definição.
- Conhecemos o conceito de *archetype patterns*; o que é e como descrevem os problemas a partir da sua essência.
- As características principais dos *archetype patterns* e as formas que podem ser utilizadas para adaptar os *archetype patterns* sem perder a essência do conceito que ele define, um recurso importante para a reusabilidade deste tipo de padrão.
- Dicas sobre padrões de codificação de código-fonte orientado a objetos.
- Conhecemos outros tipos de padrões como: DAO / DAL; BO/ BLL; DTO e MVC.

AUTOATIVIDADE

1 Seguindo o princípio Especialista da Informação, qual critério deve ser adotado na escolha de uma classe para receber uma nova responsabilidade?

2 O padrão de projeto *Factory* provê uma classe de decisão que retorna

a) um objeto de uma de suas subclasses, com base em um parâmetro recebido.
b) um atributo de uma de suas classes conexas, sem fixação de parâmetros.
c) um objeto de uma de suas subclasses, com parâmetros fatorados.
d) um atributo de uma de suas classes conexas, com base em um parâmetro reservado.

3 O padrão de projeto *singleton* é usado para restringir:

a) a instanciação de uma classe para objetos simples.
b) a instanciação de uma classe para apenas um objeto.
c) as relações entre classes e objetos.
d) classes de atributos complexos.

4 Assinale a opção que apresenta os padrões de projeto que alteram, respectivamente, a interface para um subsistema e a informação privada que será armazenada fora de um objeto:

a) *Composite* e *State*
b) *Proxy* e *Observer*
c) *Façade* e *Memento*
d) *Flyweight* e *Mediator*

UNIDADE 3

USO DE PADRÕES

OBJETIVOS DE APRENDIZAGEM

Esta unidade tem por objetivos:

- aprofundar o conhecimento em relação aos padrões de projeto;
- conhecer o processo de tomada de decisão.

PLANO DE ESTUDOS

Esta unidade de ensino contém três tópicos, sendo que no final de cada um, você encontrará atividades que contribuirão para a apropriação dos conteúdos.

TÓPICO 1 – DEFINIÇÃO, LIMITE E ESTRUTURA DE SOLUÇÃO

TÓPICO 2 – SOLUÇÃO DE PROBLEMAS ATRAVÉS DOS PADRÕES

TÓPICO 3 – PROCESSO DE TOMADA DE DECISÃO

UNIDADE 3

TÓPICO 1

DEFINIÇÃO, LIMITE E ESTRUTURA DE SOLUÇÃO

1 INTRODUÇÃO

Padrões são utilizados em todas as engenharias, inclusive a engenharia de *software*. O uso adequado de padrões nos projetos de desenvolvimento de *software* exige dedicação e conhecimento, pois não basta simplesmente conhecer o catálogo dos padrões. O ideal é colocá-los em prática. É necessário muito empenho e conhecimento para entender onde os padrões são aplicados e como devem ser utilizados em projetos de desenvolvimento de *software*.

Todo empenho dedicado na compreensão dos padrões é recompensado, pois são o suporte de projetistas de *software* na especificação da arquitetura das soluções, uma vez que as técnicas utilizadas já foram amplamente validadas. Essas técnicas criam um padrão de comunicação comum para todos os envolvidos no projeto, e transmitem a solução sem ser necessário um detalhamento mais aprofundado de todo o projeto.

2 COMO DEFINIR UMA SOLUÇÃO COM PADRÕES

Apoiar a solução de um problema na escolha de um padrão exige pensamento abstrato, pois deve-se prever que a mesma poderá/deverá ser reutilizada em aplicações diferentes.

A orientação a objetos é um importante alicerce no desenvolvimento de *software*, pois permite a extensabilidade e reusabilidade. O fato de se utilizar a orientação a objetos no desenvolvimento, não é garantia de sucesso do projeto. Uma análise dos requisitos funcionais e não funcionais é de extrema relevância para o desenvolvimento de um aplicativo com qualidade e com arquitetura flexível que permita suportar as adaptações futuras.

O histórico de projetos orientados a objetos em desenvolvimento de *software*, permite afirmar que a busca por soluções de determinados problemas possui as mesmas características de problemas que aconteceram em projetos anteriores e que por algum motivo não foram documentados. A falta de documentação acarreta em consumo de tempo e recursos financeiros para se encontrar a solução adequada, sendo que a mesma já foi explorada em situações passadas.

Equipes experientes não partem do zero para resolver os problemas de projetos. Para ganhar tempo e agilizar o desenvolvimento, reutilizam soluções testadas em projetos anteriores e os usam repetidamente. Como protagonistas desta prática, surgem os padrões de projetos, *design patterns*, que têm chamado a atenção e despertado o interesse dos projetistas de *software*, por proporcionar elementos que conduzem ao reaproveitamento de soluções para projetos, e não apenas a reutilização de código.

Como já estudamos na Unidade 2, os padrões de projetos facilitam a reutilização de soluções e arquiteturas bem-sucedidas para construir *softwares* orientados a objetos reduzindo a complexidade do processo de desenvolvimento. Além disso, o *software* orientado a objetos bem projetado possibilita aos projetistas reutilizar e empregar componentes preexistentes em sistemas futuros.

Vale lembrar que os padrões não são classes nem objeto. Eles são usados para construir conjuntos de classes e objetos.

Para a correta utilização e adaptação de padrões no projeto de desenvolvimento, é importante que se tenha conhecimento sobre:

1. Limites da solução.
2. Estrutura da solução.
3. *Frameworks* que dão suporte ao domínio.

2.1 DEFINIÇÃO DOS LIMITES DA SOLUÇÃO

É de extrema importância a delimitação da solução a ser construída. Limitar o cenário permite maior agilidade e segurança na construção, impedindo que a solução se torne redundante e equivocada.

Novamente, os requisitos se tornam protagonistas no sentido de aperfeiçoar o propósito global da solução. É importante elaborar e delimitar os eventos externos que acionam as funcionalidades das quais a solução depende.

É relevante que o projetista do projeto tenha conhecimento das situações/eventos que darão norte ao comportamento da solução. Isto permite que o especialista separe os objetos internos dos objetos externos. Este aspecto é a base para definir o limite de abrangência e usabilidade da solução proposta.

2.2 DEFINIÇÃO DA ESTRUTURA DA SOLUÇÃO

Depois de definido o limite da solução, segue-se na definição da sua estrutura que abordará conceitualmente o "espaço" da solução. A definição da estrutura define quão extensível e de fácil manutenção será o aplicativo desenvolvido.

Os requisitos também fazem parte desta etapa; nesta fase são levantados e detalhados os requisitos de sistema que devem se basear na capacidade da tecnologia de atender aos requisitos do sistema, funcionais ou não, dentro da estrutura da solução definida.

Com os requisitos do sistema usados para definir os limites da solução, existem alguns outros requisitos que podem pesar sobre a estrutura da solução: requisitos de segurança, desempenho e transacionais. Requisitos e segurança geralmente precisam do suporte de outros requisitos.

Os requisitos de desempenho têm como função permitir ou coibir a inclusão de elementos que possam sobrecarregar a estrutura da solução. Os requisitos transacionais, por sua vez, podem exigir partes adicionais da solução e que também podem de alguma forma afetar a estrutura.

2.3 COMO DEFINIR *FRAMEWORKS* DE DOMÍNIO

Frameworks são necessários para preencher os requisitos funcionais. Eles são elaborados após a definição da estrutura da solução.

Os *frameworks* permitem compreender os objetos e suas relações no alicerce do problema. Essa compreensão é a primeira etapa na criação dos *frameworks*.

> Em *Design Patterns* os padrões estão divididos em três categorias: estrutural, comportamental e de criação. Observar os requisitos da solução, sua estrutura arquitetural e os modelos de análise no contexto dessas categorias é uma boa forma de começar a definir os *frameworks* de domínio.
> FONTE: Disponível em: <https://msdn.microsoft.com/pt-br/library/bb190165.aspx>. Acesso em: 4 abr. 2016.

3 EXEMPLO DE USO DOS PADRÕES PARA DEFINIR UMA SOLUÇÃO DE *SOFTWARE*

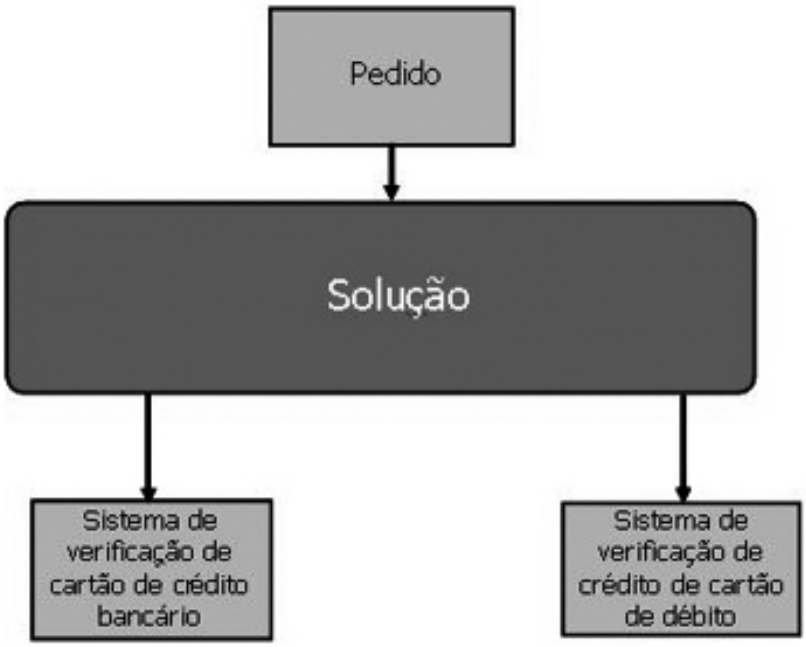
A seguir você pode entender melhor a definição do limite de solução, estrutura da solução e definição dos *frameworks* de domínio, através de um exemplo de aplicativo que faz a simulação de compra/pagamento.

3.1 LIMITES DA SOLUÇÃO

Note que o requisito da solução define que existe um pedido que tem pendência no pagamento, e que se deve validar a forma de pagamento: cartão de débito ou crédito.

A figura a seguir mostra os processos externos e os eventos que fazem a validação do tipo de pagamento.

FIGURA 29 – EVENTOS E DEPENDÊNCIAS EXTERNAS DA SOLUÇÃO

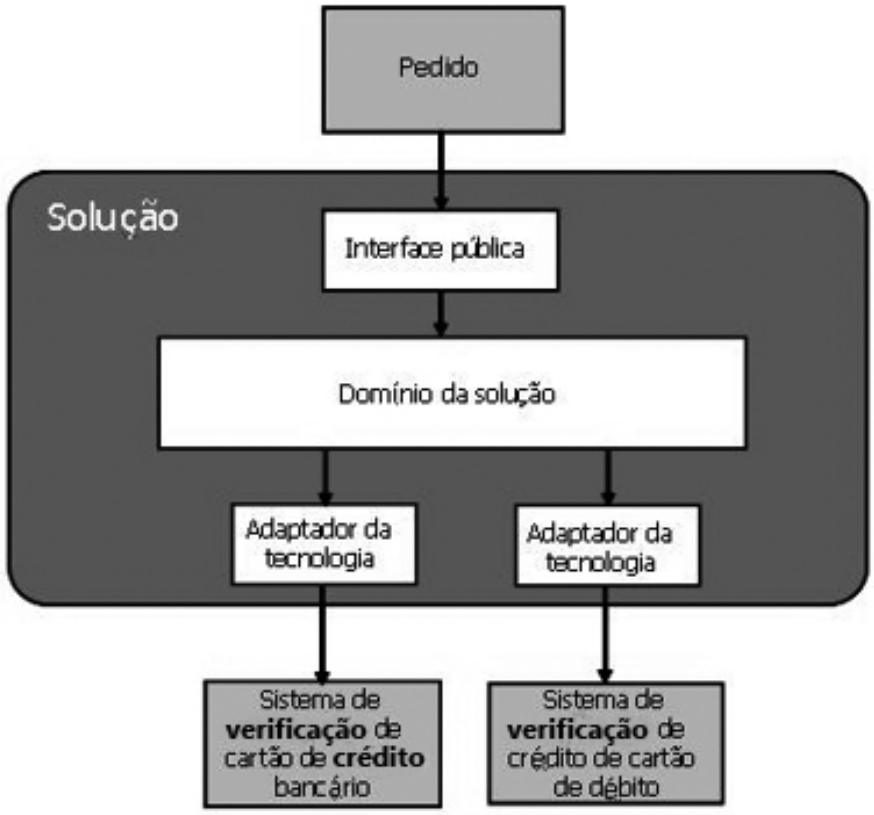


FONTE: Disponível em: <https://msdn.microsoft.com/pt-br/library/bb190165.aspx->. Acesso em: 3 mar. 2016.

3.2 ESTRUTURA DA SOLUÇÃO

O levantamento de requisitos aponta para uma interface que seja além de pública, segura. Nota-se no exemplo que a arquitetura de solução adotada é a Arquitetura em Camadas: pública, domínio e adaptadora de tecnologia. Observe na figura a seguir, a solução proposta.

FIGURA 30 – ESTRUTURA EM CAMADAS DA SOLUÇÃO



FONTE: Disponível em: <https://msdn.microsoft.com/pt-br/library/bb190165.aspx->. Acesso em: 3 mar. 2016.

3.3 MODELO DE ANÁLISE

O levantamento de requisitos da solução define que um evento externo aprovará a transação como um todo, através da validação: do número da conta, do valor da transação e a forma de pagamento (débito ou crédito). Com base nisso, pode-se concluir que existem dois objetos de domínio que respondem aos requisitos da solução: Conta e Pagamento. A figura a seguir exibe os objetos e sua relação.

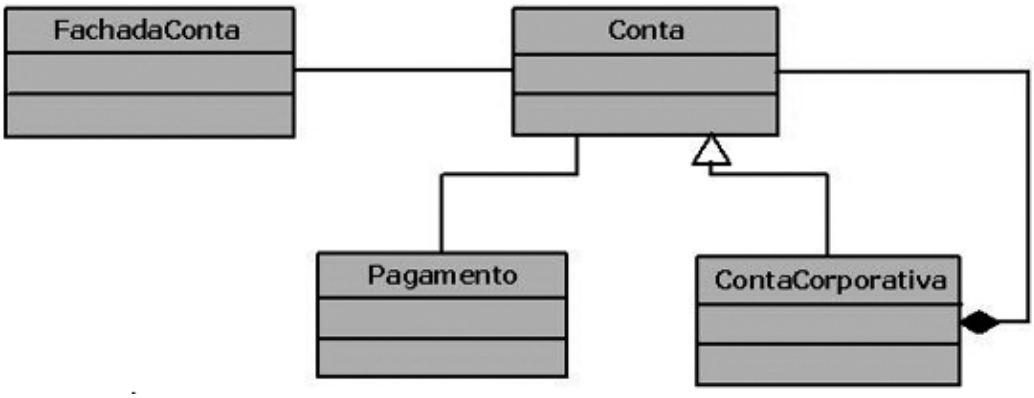
FIGURA 31 – MODELO DE ANÁLISE DA SOLUÇÃO

FONTE: Disponível em: <https://msdn.microsoft.com/pt-br/library/bb190165.aspx->. Acesso em: 4 mar. 2016.

3.4 PADRÕES ESTRUTURAIS DA SOLUÇÃO

Percebe-se pelo modelo de análise que a solução deve ser refinada, tendo como base os objetos: Conta e Pagamento. Esse refinamento leva em conta a expansão do modelo e o apontamento feito pelo próprio levantamento de requisitos, ao considerar que a conta pode ter contas filhas, no caso de se abrir uma conta em conjunto. Por causa disso, a solução implementada usa o padrão *COMPOSITION*. Para suportar aumento de tamanho e complexidade, o exemplo usa o padrão *FACADE*. A figura a seguir contempla os dois padrões propostos.

FIGURA 32 – PADRÕES ESTRUTURAIS: COMPOSIÇÃO E FACHADA



FONTE: Disponível em: <https://msdn.microsoft.com/pt-br/library/bb190165.aspx->. Acesso em: 3 mar. 2016.

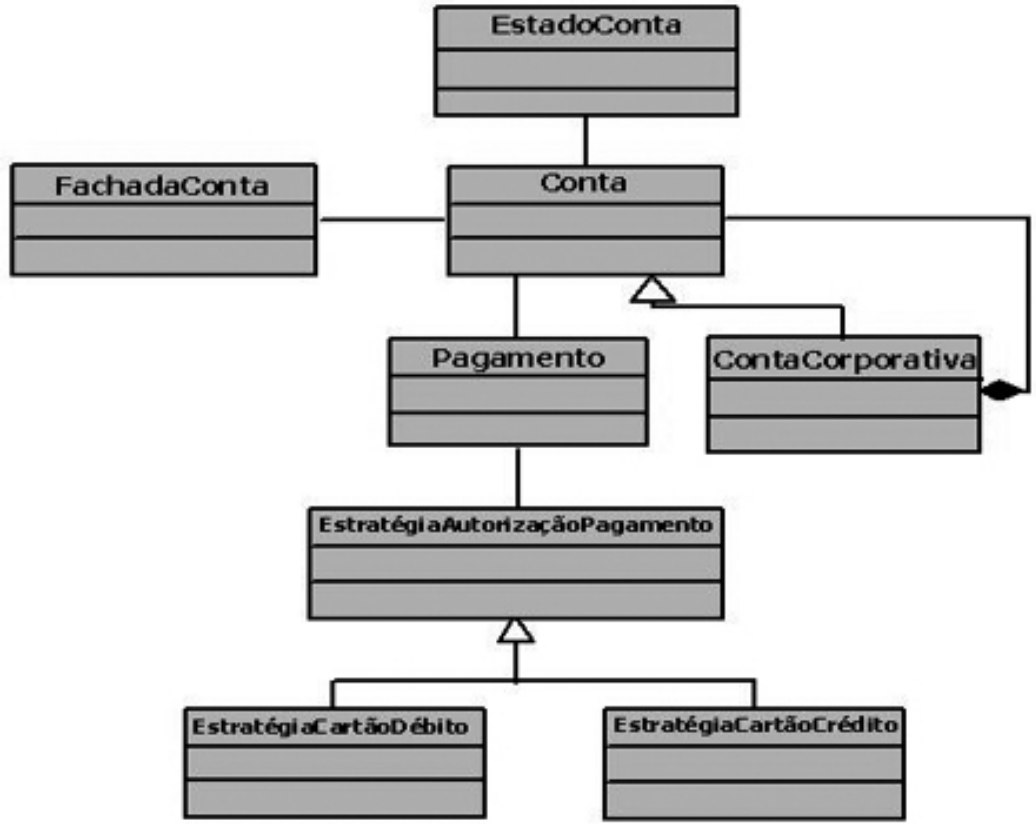
3.5 PADRÕES COMPORTAMENTAIS DA SOLUÇÃO

Através dos requisitos da solução podemos entender que a reserva de montante de valor para pagamento com cartão pode ser negada, tanto para a situação de débito, quanto de crédito. O padrão *STATE* permite associar esta restrição ao objeto *CONTA.*

Além de sinalizar o estado, é necessário que a solução tenha uma interface com dois sistemas de verificação de crédito: uma para conta corrente e outro com a operadora de cartão de crédito, sendo necessária a implementação do padrão *ESTRATEGY.*

A figura a seguir apresenta a solução com os dois padrões especificados.

FIGURA 33 – SOLUÇÃO COM PADRÕES ESTRUTURAIS E COMPORTAMENTAIS



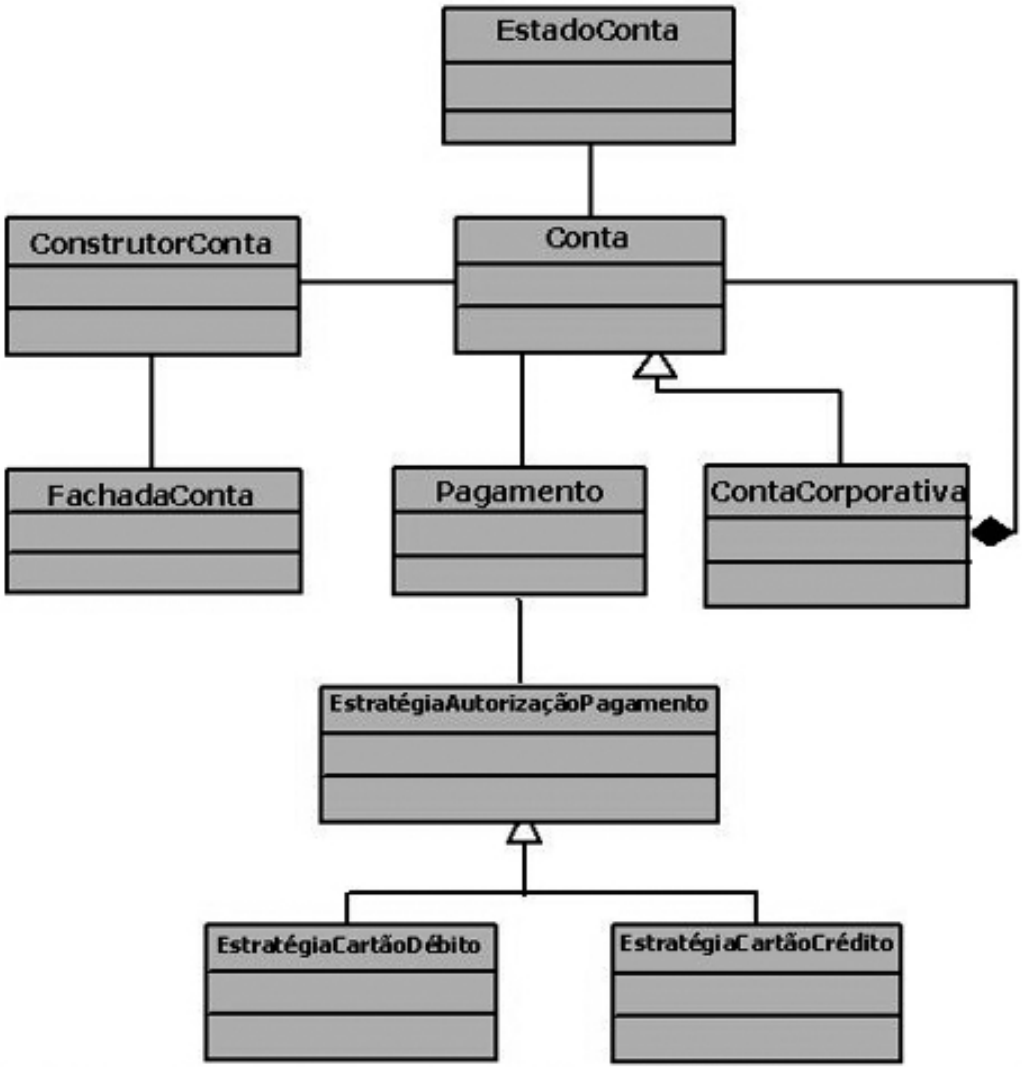
FONTE: Disponível em: <https://msdn.microsoft.com/pt-br/library/bb190165.aspx->. Acesso em: 3 mar. 2016.

3.6 PADRÕES DE CRIAÇÃO DA SOLUÇÃO

As transações de autorização do pagamento em uma conta devem ocorrer sequencialmente, e não de forma concorrente. Ou seja, apenas uma instância da conta deve estar habilitada na solução em determinado momento, sendo o último requisito a ser cumprido pela solução. Para isso, o exemplo utiliza o padrão *SINGLETON*. O padrão *CONSTRUTOR* é utilizado para encapsular a lógica *singleton* e outra lógica usada para dar vida ao objeto da conta.

A figura a seguir apresenta a estrutura de autorização do pagamento com a utilização dos padrões de criação, estrutural e comportamental.

FIGURA 34 – ESTRUTURA DE AUTORIZAÇÃO DO PAGAMENTO



FONTE: Disponível em: <https://msdn.microsoft.com/pt-br/library/bb190165.aspx->. Acesso em: 11 mar. 2016.

4 A IMPORTÂNCIA DOS PADRÕES DE PROJETO

A importância dos padrões se sustenta no fato de que são soluções amplamente testadas e aprovadas. Os padrões catalogados até o momento são muito bem definidos e documentados, tornando fácil a aplicação dos padrões na elaboração da solução dos mais variados cenários.

4.1 QUANDO OS PADRÕES NÃO O AJUDARÃO

Padrões devem ser considerados e usados como guias ou mapas, jamais como estratégia de solução. Padrões não devem ser considerados como um meio definitivo na solução. Ajudam apenas a melhorar a implementação após a definição de todos os recursos necessários ao desenvolvimento do projeto.

Antipadrão nada mais é do que fazer uso dos padrões de forma errada ou exagerada. Isso pode ser constatado pela utilização de padrões impróprios para um determinado contexto, ou uso inadequado em determinada situação.

1. Os padrões ajudam com "o que" e "como", mas não com "por que" ou "quando".
2. Existem duas noções de antipadrões:
- Aqueles que descrevem uma solução ruim para um problema que resultou em uma situação ruim;
- Aqueles que descrevem como se livrar de uma situação ruim e como proceder dessa situação para uma situação boa.

4.2 COMO SELECIONAR UM PADRÃO DE PROJETO

Com tantos padrões existentes na literatura, escolher o mais adequado para resolver um problema específico, pode se transformar em uma atividade confusa. Porém, alguns critérios podem facilitar o entendimento e escolha: considerar como os padrões de projeto solucionam problemas de projeto.

1. Avalie o que de fato o padrão faz e que tipo de problema ele soluciona.
2. Estude o relacionamento dos padrões, pois certamente você utilizará mais de um.
3. Estude os padrões semelhantes.
4. Analise possíveis causas de remodelação de projetos.
5. Verifique sempre a viabilidade do seu projeto; em vez de se focar no que pode forçar uma mudança no mesmo, avalie o que você será capaz de mudar, sem ter que remodelar a solução.

4.3 COMO USAR UM PADRÃO DE PROJETO

Depois de definir quais padrões serão utilizados na solução, faz-se necessário o seu entendimento a fim de garantir a aplicabilidade adequada. Para que isso seja possível, é necessário:

1. Ter uma visão geral do padrão, lendo-o por completo.
2. Estudar os objetos e suas relações dentro do padrão.
3. Definir nome dos participantes de forma que façam sentido dentro do projeto e padrão adotado.
4. Identificar as classes existentes na aplicação que serão afetadas pelo padrão e modificá-las.
5. Definir os nomes específicos da aplicação para as operações no padrão. Os nomes em geral dependem da aplicação. Use as responsabilidades e colaborações associadas com cada operação como guia.
6. Implementar as operações para suportar as responsabilidades e colaborações presentes do padrão.

FONTE: Adaptado de: <https://msdn.microsoft.com/pt-br/library/bb190165.aspx>. Acesso em: 9 mar. 2016.

5 PROJETAR *SOFTWARE* USANDO PADRÕES DE DESENVOLVIMENTO

Desenvolver *software* orientado a objetos reutilizável é trabalhoso. É necessário separar as classes no nível correto, definindo suas interfaces, hierarquias e as principais relações existentes. A solução deve ser específica para o problema a ser resolvido, e genérica e abstrata o suficiente, para ser replicada ou reutilizada na solução de problemas futuros e de tecnologias distintas.

Dificilmente se obtém um projeto reutilizável já na primeira vez. É preciso que uma solução seja reutilizada várias vezes, a fim de que seja aprimorada o suficiente através de modificações e melhorias que surgem em cada utilização. Boas soluções são utilizadas repetidamente. Com isso, é possível encontrar padrões de classes e de comunicações entre objetos que reaparecem com frequência em muitos projetos desenvolvidos em orientação a objetos.

Quando temos conhecimento e domínio sobre o padrão do projeto, grande parte das decisões ocorre de forma mais natural e automática, em função da carga de experiência em projeto que o próprio padrão traz. O fato de relembrar de problemas anteriores e a forma como foram solucionados é uma forma de dimensionar o tamanho da experiência adquirida em projetos anteriores.

Cada padrão explica e avalia um aspecto importante de cada projeto, tornando mais fácil reutilizar projetos e arquiteturas bem-sucedidas. A documentação e validação de técnicas largamente testadas ajudam a escolher e definir características que viabilizam a reutilização, bem como evitar situações que a comprometem. Resumindo: auxiliam a equipe na adequação dos projetos.

Nenhum padrão irá descrever um projeto novo ou que ainda não foi validado. O catálogo de padrões apresenta apenas técnicas que foram aplicadas mais de uma vez em diferentes sistemas e em cenários distintos: sistemas que usam as mesmas tecnologias e sistemas baseados em tecnologias distintas. Não há nenhum padrão que lide com concorrência ou programação distribuída, ou programação para tempo real.

Os padrões de projeto foram baseados em soluções de problemas reais implementadas nas principais linguagens de programação orientadas a objeto, como *Smalltalk* e C++. Os padrões não foram baseados em implementações em linguagens procedurais (Pascal, C, Ada) ou linguagens orientadas a objetos mais dinâmicas (CLOS, *Dylan*, *Self*).

5.1 PADRÕES DE PROJETO NO MVC DE *SMALLTALK*

O MVC (*Model*, *View* e *Controller*) é uma arquitetura ou padrão que lhe permite dividir as funcionalidades de seu sistema/*site* em camadas, essa divisão é realizada para facilitar a compreensão e solução de um problema maior.

A arquitetura MVC foi desenvolvida para ser usada em projetos de interface visual em *Smalltalk*, linguagem de programação que juntamente com o C++ ganhou grande reconhecimento na época, o MVC foi criado na década de 70, e após esses anos de sua criação ainda é um *pattern* aplicável nas mais variadas aplicações, principalmente em aplicações *web*.

O gerenciamento da interface com o usuário é uma tarefa difícil para os desenvolvedores, porém fundamental para o sucesso do sistema, além da dificuldade, a complexidade na confirmação dos dados onde necessitamos apresentar ao usuário de forma correta e ainda gerenciar a aplicação para que os usuários possam controlá-las apropriadamente, é fundamental que as interfaces e o controlador estejam sincronizados.

Há uma maneira simples de entender o MVC, uma vez que o definimos como um padrão de arquitetura, MVC é o mesmo entendimento de um *design pattern*: primeiro entendemos o problema (que é a motivação para o surgimento de um padrão), e após entender como este padrão resolve este problema, e quais as vantagens que ele nos oferece, mas com isso é necessário seguir à risca o MVC.

Quando um *software* começa a ficar grande e complexo, muitos dados são apresentados para os usuários, sentimos a necessidade de aplicar uma arquitetura que facilite nosso trabalho, desde a organização do projeto, as divisões das responsabilidades, até as possíveis modificações que poderão ser efetuadas ao longo do desenvolvimento do *software*, para isso precisaram dividir o projeto em três objetos para aplicar o MVC.

5.2 OBJETIVOS DA ARQUITETURA

MVC é composto por três tipos de objetos. O modelo é o objeto de aplicação, a vista é a apresentação na tela e o controlador define a maneira como a interface do usuário reage às entradas do mesmo. Antes do MVC, os projetos de interface para o usuário tendiam a agrupar esses objetos. MVC para aumentar a flexibilidade e a reutilização (GAMMA et al., 2000, p. 20).

O MVC tem como principal objetivo: separar dados ou lógicas de negócios (*Model*) da interface do usuário (*View*) e o fluxo da aplicação (*Controller*), a ideia é permitir que uma mensagem da lógica de negócios possa ser acessada e visualizada através de várias interfaces. Na arquitetura MVC, a lógica de negócios, ou seja, nosso *Model* não sabe quantas nem quais as interfaces com o usuário estão exibindo seu estado, a *view* não se importa de onde está recebendo os dados, mas ela tem que garantir que sua aparência reflita o estado do modelo, ou seja, sempre que os estados do modelo mudam, o modelo notifica as *view* para que as mesmas se atualizem.

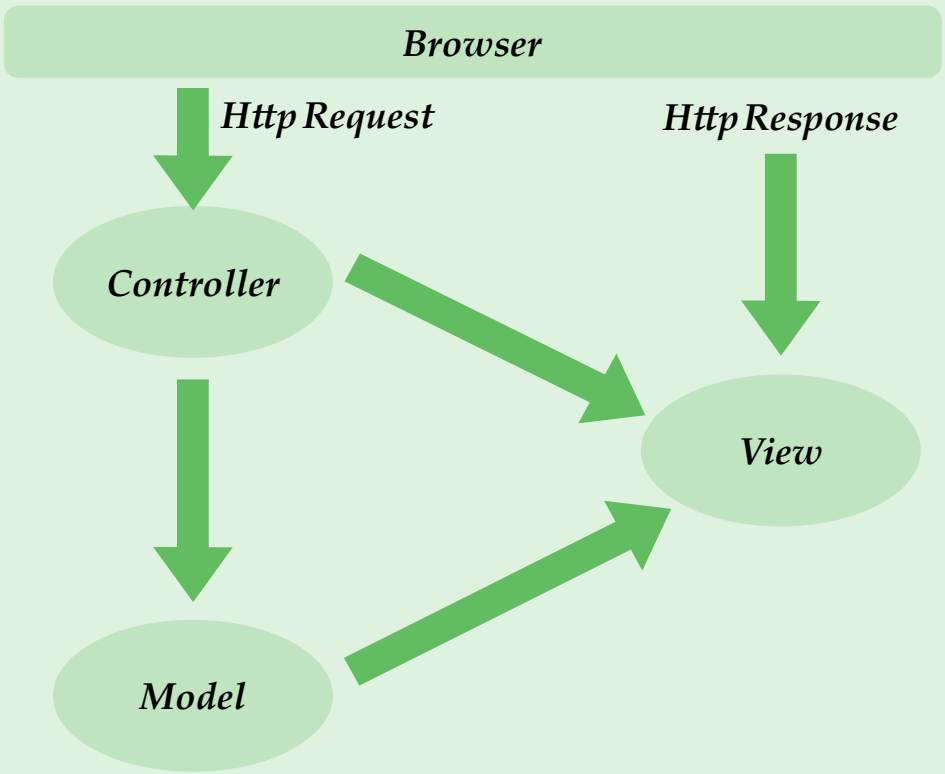
5.3 CARACTERÍSTICAS DA ARQUITETURA

- Separação entre os códigos, *View* e *Controller* que gerenciam as relações entre o *Model* e a *View*.
- Separa a lógica de negócios da apresentação.
- Possui reusabilidade, podemos criar bibliotecas e adicionar interfaces facilmente.
- É possível desenvolvê-lo em paralelo, ou seja, pode começar o projeto por qualquer camada.
- Divide as responsabilidades, ou seja, programadores na programação e *web designers* na construção visual do *software*.
- Reduz o esforço na manutenção do *software*, pois as alterações são efetuadas separadamente, não afetando as outras camadas do sistema.

5.4 ARQUITETURA

A arquitetura atua na infraestrutura dos sistemas, como comunicações, interfaces com usuários e compiladores. Agora abordaremos cada uma das camadas da arquitetura MVC, e o relacionamento com um *browser* em aplicações *web*, com os objetos *request* e *response*, onde o *request* é o pedido de uma informação ou evento, e o *response* é a resposta deste pedido à saída dos dados depois de efetuado o procedimento *request*.

FIGURA 35 – ARQUITETURA *MVC* EM APLICAÇÕES WEB



FONTE: Disponível em: <http://www.linhadecodigo.com.br/artigo/2367/abordando-a-arquitetura-mvc-e-design-patterns-observer-composite-strategy.aspx>. Acesso em: 7 mar. 2016.

MVC é um conceito (paradigma) de desenvolvimento e *design* que tenta separar uma aplicação em três partes distintas. Uma parte, a *Model*, está relacionada ao trabalho atual que a aplicação administra, outra parte, a *View*, está relacionada a exibir os dados ou informações dessa aplicação, e a terceira parte, *Controller*, em coordenar os dois anteriores exibindo a interface correta ou executando algum trabalho que a aplicação precisa completar. (GONÇALVES, 2007, p. 141).

Model ou modelo, é a camada que contém a lógica da aplicação. É a camada responsável pelas regras de negócio. Para sistemas persistentes, o modelo representa a informação (dados) dos formulários e as regras SQL para manipular dados do banco. É o modelo que mantém o estado persistente do negócio e fornece ao controlador a capacidade de acessar as funcionalidades da aplicação. O modelo somente acessa a base de dados e deixa os dados prontos para o controlador, e este, por sua vez, encaminha para a visão correta.

View, ou visão, é a camada de apresentação com o usuário, é a interface que proporcionará a entrada de dados e a visualização de respostas geradas, nas aplicações *web* é representado pelo HTML que é mostrado pelo *browser*, geralmente a visão contém formulários, tabelas, menus e botões para entrada e saída de dados. A visão deve garantir que sua apresentação reflita o estado do modelo, quando os dados do modelo mudam, o modelo notifica as vistas que dependem dele, cada vista tem a chance de atualizar-se. Desta maneira permite ligar muitas vistas a um modelo, podendo fornecer diferentes apresentações, essa camada não contém códigos relacionados à lógica de negócios, ou seja, todo o processamento é feito pelo Modelo e este repassa para a visão.

Controller**:** já descrevemos a *view* e o modelo, mas, temos de concordar que tudo isso se tornaria uma bagunça se não tivesse alguém para organizar esta arquitetura, um controlador funciona como intermediário entre a camada de apresentação e a camada de negócios, sua função, como já diz o nome, é controlar, coordenar o envio de requisições feitas entre a visão e o modelo. O *controller* define o comportamento da aplicação, o controlador é quem interpreta as solicitações (cliques, seleções de menus) feitas por usuários. Com base nestes requerimentos o controlador comunica-se com o modelo que seleciona a *view* e atualiza-a para o usuário, ou seja, o controlador controla e mapeia as ações.

Embora o MVC só contenha três camadas, há outra camada fundamental para o bom andamento da arquitetura, esta é um mecanismo de eventos necessário à comunicação entre outros três elementos, este elemento permite uma comunicação assíncrona que é invocada quando algum evento interessante acontece, esta quarta camada contém os *beans* de entidade onde se localizam os métodos *get* e *set* das classes.

FONTE: Disponível em: <http://www.linhadecodigo.com.br/artigo/2367/abordando-a-arquitetura-mvc-e-design-patterns-observer-composite-strategy.aspx#ixzz42MBGVwP8>. Acesso em: 8 mar. 2015.

6 ORGANIZANDO O CATÁLOGO

Como existem muitos padrões de projeto, é necessário classificá-los para melhor organizá-los. Os padrões são classificados em duas categorias distintas: finalidade e escopo. A finalidade indica o que o padrão faz. Pode ser finalidade de criação, de estrutura ou comportamento.

- **Padrões de criação**: se preocupam com o processo de criação dos objetos.
- **Padrões estruturais**: estão voltados para a composição das classes e dos objetos.
- **Padrões comportamentais**: analisam a interação de classes e de objetos e como é feita a distribuição de responsabilidades.

A segunda categoria de classificação (escopo) determina se o padrão se aplica a classes ou a objetos.

Padrões para classes: 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. (GAMMA et al., 2000).

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. Note que a maioria está no escopo de Objeto. (GAMMA et al., 2000).

Existem outras maneiras de organizar os padrões. Alguns padrões são frequentemente usados em conjunto. Por exemplo, o *Composite* é frequentemente usado como *Iterator* ou o *Visitor*. Alguns padrões são alternativos: o *Prototype* é frequentemente uma alternativa para o *Abstract Factory*. Alguns padrões resultam em projetos semelhantes, embora tenham intenções diferentes. Por exemplo, os diagramas de estrutura de *Composite* e *Decorator* são semelhantes.

RESUMO DO TÓPICO 1

Neste tópico você aprendeu:

- Como definir uma solução com padrões.
- Definir limite e estrutura da solução.
- Como usar um padrão de projeto.
- Projetar *software* usando Padrões de Desenvolvimento.
- Que os padrões são classificados em duas categorias distintas: finalidade e escopo.
- Que os padrões são divididos em cinco grupos: Padrões de criação; Padrões estruturais; Padrões comportamentais; Padrões para classes; e Padrões para objetos.
- Nenhum padrão irá descrever um projeto novo ou que ainda não foi validado.
- Que a solução deve ser específica para o problema a ser resolvido, e genérica e abstrata o suficiente, para ser replicada ou reutilizada na solução de problemas futuros e de tecnologias distintas.
- Que padrões devem ser considerados e usados como guias ou mapas, jamais como estratégia de solução.
- Que padrões não devem ser considerados como um meio definitivo na solução.
- Que os padrões ajudam apenas a melhorar a implementação após a definição de todos os recursos necessários ao desenvolvimento do projeto.
- Que antipadrão nada mais é do que fazer uso dos padrões de forma errada ou exagerada. Isso pode ser constatado pela utilização de padrões impróprios para um determinado contexto, ou uso inadequado em determinada situação.

AUTOATIVIDADE

1 O que é um padrão de projeto?

2 Indique pelo menos três vantagens de usar padrões de projeto.

3 Como os padrões de projeto GOF são apresentados?

4 Quais são as categorias dos padrões de projetos?

TÓPICO 2

SOLUÇÃO DE PROBLEMAS ATRAVÉS DOS PADRÕES

1 INTRODUÇÃO

Neste tópico você terá uma visão mais aprofundada acerca dos problemas que os padrões de projetos ajudam a solucionar.

2 COMO OS PADRÕES SOLUCIONAM PROBLEMAS DE PROJETO

A etapa inicial é a procura por objetos apropriados: a base da construção orientada a objetos é o **objeto**. Este considera os dados, também referenciados como **métodos** ou **operações**. Somente através de uma mensagem recebida, o objeto executa uma operação. Por causa disso, o estado interno do objeto é conhecido como **encapsulamento.** Isso garante que ele não seja acessado diretamente, tornando invisível sua representação no exterior.

Muitos conceitos tornam difícil a fatoração de um projeto orientado a objetos: encapsulamento, granularidade, dependência, flexibilidade, desempenho, evolução, reutilização etc. Todos influenciam o particionamento da construção; geralmente de formas distintas e conflitantes.

Existem diferentes abordagens para apoiar um projeto orientado a objetos. As metodologias são distintas e permitem descrever meticulosamente um problema, mapear as colaborações e responsabilidades do sistema ou modelar cenários reais transformando-os em objetos após efetuada a análise.

Boa parte dos objetos do projeto tem origem no modelo de análise. Outros objetos surgem e têm classes que não tem identificação no cenário analisado. A modelagem restrita do mundo real reflete apenas a situação atual, mas não contempla a realidade futura. As abstrações que surgem durante um projeto são as chaves para torná-lo flexível no futuro.

Neste sentido, os padrões de projeto ajudam a identificar abstrações que são mais difíceis de visualizar. Processos e códigos de programação não são objetos do mundo real, porém, são de extrema importância para o desenvolvimento do projeto. O padrão *Strategy* descreve como implementar famílias de código de

programação intercambiáveis. O padrão *State* representa cada estado de uma entidade como um objeto. Esses objetos são raramente encontrados durante a análise, ou mesmo durante os estágios iniciais de um projeto; eles são descobertos mais tarde quando se estuda a reutilização e flexibilização do projeto.

2.1 DETERMINAR A GRANULARIDADE DOS OBJETOS

É complexo determinar tamanho e número de objetos. Novamente, os padrões servem de auxílio. Alguns exemplos:

- **Padrão *Facade***: descreve como representar subsistemas completos, na forma de objetos.
- **Padrão *Flyweight***: descreve como suportar uma quantidade maior de objetos.
- **Padrão *Abstract Factory* e *Builder***: fornecem objetos cuja função é de suportar outros objetos.
- **Padrão *Visitor* e *Command***: fornecem objetos para implementar/executar uma operação/solicitação em outro objeto ou grupo de objetos.

2.2 ESPECIFICAR INTERFACES DE OBJETOS

Cada operação declarada por um objeto especifica o nome da operação, os objetos que ela aceita como parâmetros e o valor retornado por ela. Isso é conhecido como **assinatura** da operação. O conjunto de todas as assinaturas definido pelas operações de um objeto é chamado de **interface** do objeto. A interface de um objeto caracteriza o conjunto completo de solicitações que lhe podem ser enviadas. Qualquer solicitação que corresponde a uma assinatura na interface do objeto pode ser enviada para o mesmo.

Um **tipo** é um nome usado para denotar uma interface específica. Quando dizemos que um objeto tem o tipo "Janela", significa que ele aceita todas as solicitações para as operações definidas na interface chamada "Janela". Um objeto pode ter muitos tipos, assim como objetos muito diferentes podem compartilhar um mesmo tipo. Parte da interface de um objeto pode ser caracterizada por um tipo, e outras partes por outros tipos. Dois objetos do mesmo tipo necessitam compartilhar somente partes de suas interfaces. As interfaces podem conter outras interfaces como subconjuntos.

Dizemos que um tipo é um **subtipo** de outro se sua interface contém a interface do seu **supertipo**. Frequentemente dizemos que um subtipo *herda* a interface do seu supertipo.

As interfaces são fundamentais em sistemas orientados a objetos. Os objetos são conhecidos somente através das suas interfaces. Não existe nenhuma maneira de saber algo sobre um objeto ou de pedir que faça algo sem intermédio de sua interface. A interface de um objeto nada diz sobre sua implementação – diferentes objetos estão livres para implementar as solicitações de diferentes maneiras. Isso significa que dois objetos que tenham implementações completamente diferentes podem ter interfaces idênticas.

Quando uma mensagem é enviada a um objeto, a operação específica que será executada depende de ambos – mensagem e objeto receptor. Diferentes objetos que suportam solicitações idênticas, podem ter diferentes implementações das operações que atendem a estas solicitações. A associação em tempo de execução de uma solicitação a um objeto e a uma das suas operações, é conhecida como **ligação dinâmica (*dynamic binding*).**

O uso da ligação dinâmica significa que o envio de uma solicitação não o prenderá a uma implementação particular até o momento da execução. Consequentemente, você poderá escrever programas que esperam um objeto com uma interface em particular, sabendo que qualquer objeto que tenha a interface correta aceitará a solicitação. Além do mais, a ligação dinâmica permite substituir uns pelos outros objetos que tenham interfaces idênticas. Essa capacidade de substituição é conhecida como **polimorfismo** e é um conceito-chave em sistemas orientados a objetos. Ela permite a um objeto-cliente criar poucas hipóteses sobre outros objetos, exceto que eles suportam uma interface específica. O polimorfismo simplifica as definições dos clientes, desacopla objetos entre si e permite a eles variarem seus inter-relacionamentos em tempo de execução.

Os padrões de projeto ajudam a definir interfaces pela identificação de seus elementos e pelos tipos de dados que são enviados através de uma interface. Um padrão de projeto também pode lhe dizer o que não colocar na interface. O padrão Memento é um bom exemplo. Ele descreve como encapsular e salvar o estado interno de um objeto de modo que o objeto possa ser restaurado àquele estado mais tarde. O padrão estipula que objetos Memento devem definir duas interfaces: uma restrita, que permite aos clientes manterem e copiarem mementos, e uma privilegiada, que somente o objeto original pode usar para armazenar e recuperar estados no Memento.

Os padrões de projeto também especificam relacionamentos entre interfaces. Em particular, frequentemente exigem que algumas classes tenham interfaces similares, ou colocam restrições sobre interfaces de

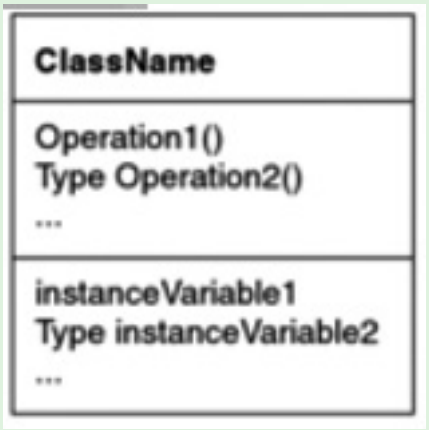
algumas classes. Por exemplo, tanto *Decorator* quanto *Proxy* exigem que as interfaces de objetos *Decorator* como *Proxy* sejam idênticas aos objetos "decorados" e "representados". Em *Visitor*, a interface de *Visitor* deve refletir todas as classes de objetos que *visitors* (visitantes) podem visitar.

2.3 ESPECIFICAR IMPLEMENTAÇÕES DE OBJETOS

Até aqui dissemos pouco sobre como efetivamente definimos um objeto. Uma implementação de um objeto é definida por sua **classe**. A classe especifica os dados internos do objeto e de sua representação e define as operações que o objeto pode executar.

Nossa anotação, baseada na OMT, ilustra uma classe como um retângulo com o nome da classe em negrito. As operações aparecem em tipo normal abaixo do nome da classe. Quaisquer dados que a classe defina vêm em seguida às operações. O nome da classe é separado das operações por linhas, da mesma forma que as operações dos dados:

FIGURA 36 – IMPLEMENTAÇÕES DE OBJETOS



FONTE: Gamma et al. (2000, p. 30).

Tipos de Retorno e tipos de variáveis de instância são opcionais, uma vez que não assumimos uma linguagem de implementação estaticamente tipificada.

Os objetos são criados por **instanciação** de uma classe. Diz-se que o objeto é uma **instância** da classe. O processo de instanciar uma classe aloca memória para os dados internos do objeto (compostos de **variáveis de**

instância) e associa as operações a estes dados. Muitas instâncias semelhantes de um objeto podem ser criadas pela instanciação de uma classe.

Uma flecha tracejada indica uma classe que instancia objetos de outra classe. A flecha aponta para a classe dos objetos instanciados. (GAMMA et al., 2000, p. 29-30).

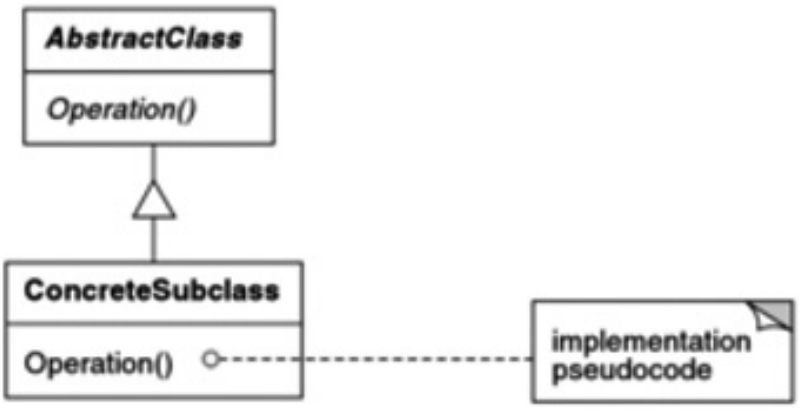
2.4 INSTANCIADOR INSTANCIADO

A Herança existe para permitir definir classes a partir de classes que já existem. As novas classes herdam todas as definições de dados e operações da classe mãe.

As **classes abstratas** existem para definir interfaces comuns para as suas próprias subclasses. Vale lembrar que classes abstratas não podem ser instanciadas. Classes abstratas apenas declaram operações abstratas, sem executá-las.

Na maioria das vezes as subclasses aperfeiçoam os comportamentos de suas classes de origem, e os diagramas colaboram com isso na medida que permitem incluir pseudocódigos para implementação de uma operação; neste caso, o código aparecerá em uma caixa com um canto dobrado, conectada por uma linha pontilhada à operação que ele implementa.

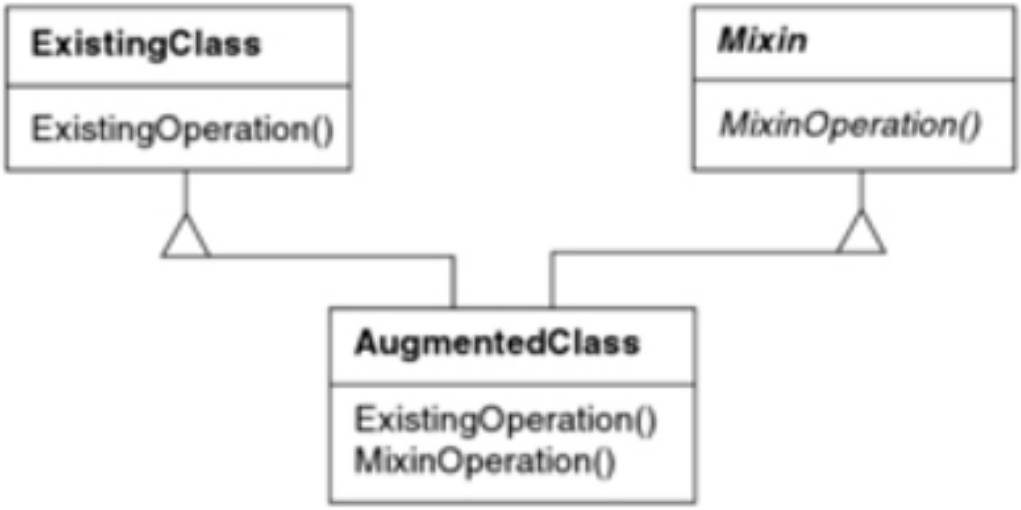
FIGURA 37 – CLASSES ABSTRATAS



FONTE: Gamma et al. (2000, p. 31)

Existem ainda as classes **mixin,** que têm o intuito de oferecer funcionalidades opcionais para outras classes. Estas classes são semelhantes às classes abstratas pois não podem ser instanciadas. E, ainda, exigem herança múltipla.

FIGURA 38 – CLASSE MIXIN



FONTE: Gamma et al. (2000, p. 31)

2.5 HERANÇA DE CLASSE *VERSUS* HERANÇA DE INTERFACE

É importante compreender a diferença entre a *classe* de um objeto e seu tipo. A classe de um objeto define como ele é implementado. A classe define o estado interno do objeto e a implementação de suas operações. Em contraste a isso, o tipo de um objeto se refere somente à sua interface – o conjunto de solicitações às quais ele pode responder. Um objeto pode ter muitos tipos, e objetos de diferentes classes podem ter o mesmo tipo.

Naturalmente, existe um forte relacionamento entre classe e tipo. Uma vez que uma classe define as operações que um objeto pode executar, ela também define o tipo do objeto. Quando dizemos que o objeto é uma instância de uma classe, queremos dizer que o objeto suporta a interface definida pela classe.

Linguagens como C++ e *Eiffel* utilizam classes para especificar tanto o tipo de um objeto como sua implementação. Os programas em *Smalltalk* não declaram os tipos de variáveis; consequentemente, o compilador não verifica se os tipos dos objetos atribuídos a uma variável são subtipos do tipo da variável. Enviar uma mensagem exige a verificação de que a classe do receptor implementa a mensagem, mas não exige a verificação de que o receptor seja uma instância de uma classe específica.

É também importante compreender a diferença entre herança de classe e herança de interface (ou subtipificação). A herança de classe define a implementação de um objeto em termos da implementação de outro objeto. Resumidamente, é um mecanismo para compartilhamento de código e de representação. Diferentemente disso, a herança de interface (ou subtipificação) descreve quando um objeto pode ser usado no lugar de outro.

É fácil confundir esses dois conceitos porque muitas linguagens não fazem uma distinção explícita. Em linguagens como C++ e *Eiffel*, herança significa tanto herança de interface como de implementação. A maneira-padrão de herdar uma interface em C++ é herdar publicamente de uma classe que tem apenas funções-membro virtuais.

Herança pura de interface assemelha-se em C++ a herdar publicamente de classes abstratas puras. A herança pura de implementação, ou herança de classe, pode ser assemelhada com a herança privada. Em *Smalltalk*, herança significa somente herança de implementação. Você pode atribuir instâncias de qualquer classe a uma variável, contanto que essas instâncias apoiem a operação executada sobre o valor da variável.

Embora muitas linguagens de programação não apoiem a distinção entre herança de interface e de implementação, as pessoas fazem a distinção na prática. Os programadores *Smalltalk* usualmente pensam como se as subclasses fossem subtipos (embora existam algumas exceções bem conhecidas); programadores C++ manipulam objetos através de tipos definidos por classes abstratas.

Muitos dos padrões de projeto dependem desta distinção. Por exemplo, os objetos numa *Chain of Responsibility* devem ter um tipo em comum, mas usualmente não compartilham uma implementação. No padrão *Composite*, o *Component* define uma interface comum, porém *Composite* frequentemente define uma implementação em comum. O *Command*, o *Observer*, o *State* e o *Strategy* são frequentemente implementados com classes abstratas que são puramente interfaces. (GAMMA et al., 2000, p. 31-33).

2.6 PROGRAMANDO PARA UMA INTERFACE, NÃO PARA UMA IMPLEMENTAÇÃO

A herança permite ampliar as funcionalidades de uma aplicação pois viabiliza a reutilização das funcionalidades das classes de origem. Permite criar um objeto de forma mais rápida, pois se baseia em objetos já existentes.

Tendo isso em mente, não declare variáveis como instâncias de classes concretas específicas. Prefira interfaces definidas por uma classe abstrata, que é tema comum para os padrões de projeto.

Certamente você terá de instanciar classes concretas no seu aplicativo, e os padrões de criação prestam grande auxílio neste sentido:

Abstract Factory, *Builder*, *Factory Method*, *Prototype* e *Singleton*: esses padrões fornecem maneiras diferentes de associar uma interface com a implementação, quando a mesma for instanciada. Estes padrões são a garantia de que o aplicativo será escrito com foco para a interface e não para a implementação.

2.7 COLOCANDO OS MECANISMOS DE REUTILIZAÇÃO PARA FUNCIONAR

"Muitas pessoas podem compreender conceitos como objetos, interfaces, classes e herança. O desafio reside em aplicá-los à construção de *software* flexível e reutilizável, e os padrões de projeto podem mostrar como fazê-lo". (GAMMA et al., 2000, p. 33).

2.8 HERANÇA VERSUS COMPOSIÇÃO

As duas técnicas mais comuns para a reutilização de funcionalidade em sistemas orientados a objetos são herança de classe e **composição de objetos**. Como já explicamos, a herança de classe permite definir a implementação de uma classe em termos da implementação de outra. A reutilização por meio de subclasses é frequentemente chamada de **reutilização de caixa branca** (ou aberta). O termo "caixa branca" se refere à visibilidade: com herança, os interiores das classes ancestrais são frequentemente visíveis para as subclasses.

A composição de objetos é uma alternativa à herança de classe. Aqui, a nova funcionalidade é obtida pela montagem e/ou composição de objetos, para obter funcionalidades mais complexas. A composição de objetos requer que os objetos que estão sendo compostos tenham interfaces bem definidas. Esse estilo de reutilização é chamado **reutilização de caixa preta,** porque os detalhes internos dos objetos não são visíveis. Os objetos aparecem somente como "caixas pretas".

A herança e a composição têm, cada uma, vantagens e desvantagens. A herança de classes é definida estaticamente em tempo de compilação e é simples de usar, uma vez que é suportada diretamente pela linguagem de programação. A herança de classe também torna mais fácil modificar a implementação que está sendo reutilizada.

Quando uma subclasse redefine algumas, mas não todas as operações, ela também pode afetar as operações que herda, assumindo-se

que elas chamam as operações redefinidas. Porém, a herança de classe tem também algumas desvantagens. Em primeiro lugar, você não pode mudar as implementações herdadas das classes ancestrais em tempo de execução, porque a herança é definida em tempo de compilação.

Em segundo lugar, e geralmente isso é o pior, as classes ancestrais frequentemente definem pelo menos parte da representação física das suas subclasses. Porque a herança expõe para uma subclasse os detalhes da implementação dos seus ancestrais, frequentemente é dito que "a herança viola o encapsulamento". A implementação de uma subclasse, dessa forma, torna-se tão amarrada à implementação da sua classe que qualquer mudança na implementação desta forçará uma mudança naquela.

As dependências de implementação podem causar problemas quando se está tentando reutilizar uma subclasse. Se algum aspecto da implementação herdada não for apropriado a novos domínios de problemas, a classe-mãe deve ser reescrita ou substituída por algo mais apropriado. Esta dependência limita a flexibilidade e, em última instância, a reusabilidade. Uma cura para isto é herdar somente de classes abstratas, uma vez que elas normalmente fornecem pouca ou nenhuma implementação.

A composição de objetos é definida dinamicamente em tempo de execução pela obtenção de referências a outros objetos através de um determinado objeto. A composição requer que os objetos respeitem as interfaces uns dos outros, o que por sua vez exige interfaces cuidadosamente projetadas, que não impeçam você de usar um objeto com muitos outros. Porém, existe um ganho. Como os objetos são acessados exclusivamente através de suas interfaces, nós não violamos o encapsulamento.

Qualquer objeto pode ser substituído por outro em tempo de execução, contanto que tenha o mesmo tipo. Além do mais, como a implementação de um objeto será escrita em termos de interfaces de objetos, existirão substancialmente menos dependências de implementação.

A composição de objetos tem um outro efeito sobre o projeto de um sistema. Dar preferência à composição de objetos à herança de classes ajuda a manter cada classe encapsulada e focalizada em uma única tarefa. Suas classes e hierarquias de classes se manterão pequenas, com menor probabilidade de crescerem até se tornarem monstros intratáveis. Por outro lado, um projeto baseado na composição de objetos terá mais objetos (embora menos classes), e o comportamento do sistema dependerá de seus inter-relacionamentos ao invés de ser definido em uma classe.

Isto nos conduz ao nosso segundo princípio de projeto orientado a objetos: prefira a composição de objeto à herança de classe. Idealmente, você não deveria ter que criar novos componentes para obter reutilização.

Deveria ser capaz de conseguir toda a funcionalidade de que necessita simplesmente montando componentes existentes através da composição de objetos. Mas este raramente é o caso, porque o conjunto de componentes disponíveis nunca é exatamente rico o bastante na prática. A reutilização por herança torna mais fácil criar novos componentes que podem ser obtidos pela composição de componentes existentes. Assim, a herança e a composição de objetos trabalham juntas.

No entanto, nossa experiência mostra que os projetistas abusam da herança como uma técnica de reutilização, e que frequentemente os projetos tornam-se mais reutilizáveis (e mais simples) ao preferir a composição de objetos. Você verá a composição de objetos aplicada repetidas vezes nos padrões de projeto. (GAMMA et al., 2000, p. 33-35).

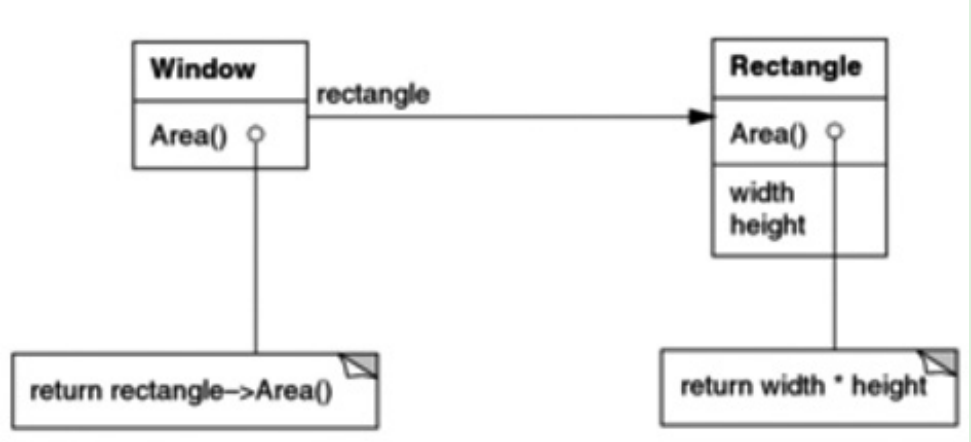
2.9 DELEGAÇÃO

Delegação é uma maneira de tornar a composição tão poderosa para fins de reutilização quanto à herança. Na delegação, dois objetos são envolvidos no tratamento de uma solicitação: um objeto receptor delega operações para o seu **delegado**; isto é análogo à postergação de solicitações enviadas às subclasses para as suas classes-mãe. Porém, com a herança, uma operação herdada pode sempre se referir ao objeto receptor através da variável membro *this*, em C++ e *self* em *Smalltalk*. Para obter o mesmo efeito com o uso de delegação, o receptor passa a si mesmo para o delegado para permitir à operação delegada referenciar o receptor.

Por exemplo, em vez de fazer da classe *Window* uma subclasse de *Rectangle* (porque janelas são retangulares), a classe *Window* deve reutilizar o comportamento de *Rectangle*, conservando uma variável de instância de *Rectangle*, e *delegando* o comportamento específico de *Rectangle* para ela. Em outras palavras, ao invés de uma *Window* **ser** um *Rectangle* ela **teria** um *Rectangle*. Agora, *Window* deve encaminhar as solicitações para sua instância *Rectangle* explicitamente, ao passo que antes ela teria herdado essas operações.

O seguinte diagrama ilustra a classe *Window* delegando sua operação área a uma instância de *Rectangle*.

FIGURA 39 – CLASSE WINDOW



FONTE: Gamma et al. (2000, p. 34)

Uma flecha com uma linha cheia indica que um objeto de uma classe mantém uma referência para uma instância de outra classe. A referência tem um nome opcional, nesse caso, "*rectangle*".

A principal vantagem da delegação é que ela torna fácil compor comportamentos sem tempo de execução e mudar a forma como são compostos. A nossa *Window* pode se tornar circular em tempo de execução, simplesmente pela substituição da sua instância *Rectangle* por uma instância de *Circle*, assumindo-se que *Rectangle* e *Circle* tenham o mesmo tipo.

A delegação tem uma desvantagem que ela compartilha com outras técnicas que tornam o *software* mais flexível através da composição de objetos: o *software* dinâmico, altamente parametrizado, é mais difícil de compreender do que o *software* mais estático. Há também ineficiências de tempo de execução, mas as ineficiências humanas são mais importantes a longo prazo. A delegação é uma boa escolha de projeto somente quando ela simplifica mais do que complica. Não é fácil estabelecer regras que lhe digam exatamente quando usar delegação, porque o quão efetiva ela será dependerá das condições do contexto e de quanta experiência você tem com o seu uso. A delegação funciona melhor quando é usada em formas altamente estilizadas, isto é, em padrões catalogados.

Diversos padrões de projeto usam delegação. Os padrões *State*, *Strategy* e *Visitor* dependem dela. No padrão *State*, um objeto delega solicitações para um objeto *State* que representa o seu estado atual. No padrão *Strategy*, um objeto delega uma solicitação específica para um objeto que representa uma estratégia para executar a solicitação. Um objeto terá somente um estado, mas ele pode ter muitas estratégias para diferentes

solicitações. A finalidade de ambos os padrões é mudar o comportamento de um objeto pela mudança dos objetos para os quais ele delega solicitações. Em *Visitor*, a operação que é executada em cada elemento da estrutura de um objeto é sempre delegada para o objeto *Visitor*.

Outros padrões utilizam delegação menos intensamente. O *Mediator* introduz um objeto para intermediar a comunicação entre outros objetos. Algumas vezes, o objeto Mediador simplesmente implementa operações passando-as adiante para outros objetos; outras vezes, ele passa junto uma referência para si próprio, e assim usa a delegação no seu sentido exato. O *Chain of Responsibility* trata as solicitações passando-as para frente, de um objeto para o outro, ao longo de uma cadeia de objetos. Algumas vezes essa solicitação carrega consigo uma referência ao objeto original que recebeu a solicitação, e nesse caso o padrão está usando delegação.

O *Bridge* desacopla uma abstração de sua implementação. Se a abstração e uma particular implementação são muito parecidas, então a abstração pode simplesmente delegar operações para aquela implementação.

A delegação é um exemplo extremo da composição de objetos. Ela mostra que você pode sempre substituir a herança pela composição de objetos como um mecanismo para a reutilização de código. (GAMMA et al., 2000, p. 35-37).

2.10 RELACIONANDO ESTRUTURAS DE TEMPO DE EXECUÇÃO E DE TEMPO DE COMPILAÇÃO

A estrutura em tempo de execução de um programa orientado a objetos é bem diferente de sua estrutura quando pensamos no código da implementação. A estrutura de um aplicativo em tempo de execução deve ser modelada muito mais pela percepção do projetista do que pela linguagem de implementação. O projetista deve tomar cuidado com os objetos (seus tipos e relacionamentos), pois estes sim, são determinantes para uma boa estrutura de execução do aplicativo.

Os padrões de projeto que capturam o escopo dos objetos, são eficientes ao distinguir as estruturas de tempo de compilação e execução.

- *Composite* e *Decorator*: facilitam a construção de estruturas complexas de tempo de execução.
- *Observer*: facilitam a compreensão de estruturas de tempo de execução mais complexas.
- *Chain of Responsibility*: fornecem padrões de comunicação que a herança não revela.

2.11 PROJETANDO PARA MUDANÇAS

A chave para maximização da reutilização está na antecipação de novos requisitos e mudanças nos requisitos existentes e em projetar sistemas de modo que eles possam evoluir de acordo.

Para projetar o sistema de maneira que seja robusto face a tais mudanças, você deve levar em conta como o sistema pode necessitar mudar ao longo de sua vida. Um projeto que não leva em consideração a possibilidade de mudanças está sujeito ao risco de uma grande reformulação no futuro. Essas mudanças podem envolver redefinições e reimplementações de classes, modificação de clientes e retestagem do sistema. A reformulação afeta muitas partes de um sistema de *software* e, invariavelmente, mudanças não-antecipadas são caras.

Os padrões de projeto ajudam a evitar esses problemas ao garantirem que o sistema possa mudar segundo maneiras específicas. Cada padrão de projeto permite a algum aspecto da estrutura do sistema variar independentemente de outros aspectos, desta forma tornando um sistema mais robusto em relação a um tipo particular de mudança.

Aqui apresentamos algumas causas comuns de reformulação de projeto, junto com o(s) padrão(ões) que as tratam:

1. **Criando um objeto pela especificação explícita de uma classe**. Especificar um nome de uma classe quando você cria um objeto faz com que se comprometa com uma implementação em particular, em vez de se comprometer com uma determinada interface. Este compromisso pode complicar futuras mudanças. Para evitá-lo, crie objetos indiretamente. Padrões de projeto: *Abstract Factory*, *Factory Method*, *Prototype*.

2. **Dependência de operações específicas**. Quando você especifica uma operação em particular, se compromete com uma determinada maneira de atender a uma solicitação. Evitando solicitações codificadas inflexivelmente (*hard-coded*), você torna mais fácil mudar a maneira como uma solicitação é atendida, tanto em tempo de compilação como em tempo de execução. Padrões de projeto: *Chain of Responsibility*, *Command*.

3. **Dependência da plataforma de *hardware* e *software***. As interfaces externas do sistema operacional e as interfaces de programação de aplicações (APIs) são diferentes para diferentes plataformas de *hardware* e *software*. O *software* que depende de uma plataforma específica será mais difícil de portar para outras plataformas. Pode ser até mesmo difícil mantê-

lo atualizado na sua plataforma nativa. Portanto, é importante projetar o seu sistema para a limitar suas dependências de plataformas. Padrões de projeto: *Abstract Factory*, *Bridge*.

4. **Dependência de representações ou implementações de objetos**. As classes clientes podem ser alteradas quando ocorrer mudanças nos objetos. Padrões de projeto: *Abstract Factory*, *Bridge*, *Memento*, *Proxy*.

5. **Dependências algorítmicas**. Os algoritmos são frequentemente estendidos, otimizados e substituídos durante desenvolvimento e reutilização. Os objetos que dependem de algoritmos terão que mudar quando o algoritmo mudar. Portanto os algoritmos que provavelmente mudarão deveriam ser isolados. Padrões de projeto: *Builder*, *Iterator*, *Strategy*, *Template Method*, *Visitor*.

6. **Acoplamento forte**. Classes que são fortemente acopladas são difíceis de reutilizar isoladamente, uma vez que dependem umas das outras. O acoplamento forte leva a sistemas monolíticos, nos quais você não pode mudar ou remover uma classe sem compreender e mudar muitas outras classes. O sistema torna-se uma massa densa difícil de aprender, portar e manter. Um acoplamento fraco aumenta a probabilidade de que uma classe possa ser usada por si mesma e de que um sistema possa ser aprendido, portado, modificado e estendido mais facilmente. Os padrões de projeto usam técnicas como acoplamento abstrato e projeto em camadas para obter sistemas fracamente acoplados. Padrões de projeto: *Abstract Factory*, *Bridge*, *Chain of Responsibility*, *Command*, *Façade*, *Mediator*, *Observer*.

7. **Estendendo a funcionalidade pelo uso de subclasses**. Customizar ou adaptar um objeto através do uso de subclasses não costuma ser fácil. Cada classe nova tem um custo adicional (*overhead*) de inicialização, finalização etc. Definir uma subclasse exige uma compreensão profunda da classe-mãe. Por exemplo, redefinir uma operação pode exigir a redefinição de outra (em outro lugar do código). Uma operação redefinida pode ser necessária para chamar uma operação herdada. E o uso de subclasses pode levar a uma explosão de classes, porque você pode ter que introduzir muitas subclasses novas, até mesmo para uma extensão simples. A composição de objetos, em geral, e a delegação, em particular, fornecem alternativas flexíveis à herança para a combinação de comportamentos. Novas funcionalidades podem ser acrescentadas a uma aplicação pela composição de objetos existentes de novas maneiras, em vez de definir novas subclasses a partir das classes existentes. Por outro lado, o uso intenso da composição de objetos pode tornar os projetos menos compreensíveis. Muitos padrões de projeto produzem arquiteturas (*designs*) nas quais você pode introduzir uma

funcionalidade customizada simplesmente pela definição de uma subclasse e pela composição de suas instâncias com as existentes. Padrões de projeto: *Bridge*, *Chain of Responsibility*, *Composite*, *Decorator*, *Observer*, *Strategy*.

8. **Incapacidade para alterar classes de modo conveniente**. Algumas vezes você tem que modificar uma classe que não pode ser convenientemente modificada. Talvez necessite do código-fonte e não disponha do mesmo (como pode ser o caso em se tratando de uma biblioteca comercial de classes). Ou, talvez, qualquer mudança possa requerer a modificação de muitas subclasses existentes. Padrões de projeto oferecem maneiras para modificações de classes em tais circunstâncias. Padrões de projeto: *Adapter*, *Decorator*, *Visitor*.

Estes exemplos refletem a flexibilidade que os padrões de projeto podem ajudá-lo a incorporar ao seu *software*. Quão crucial é tal flexibilidade? Depende do tipo de *software* que você está construindo.

Se você está construindo um programa de aplicação tal como um editor de documentos ou uma planilha, então as prioridades mais altas são reutilizabilidade interna, facilidade de manutenção e de extensão. A reutilizabilidade interna garante que você não projete, nem implemente, mais do que necessita.

Os padrões de projeto que reduzem dependências podem aumentar a reusabilidade interna. O acoplamento mais fraco aumenta a probabilidade de que uma classe de objetos possa cooperar com várias outras. Por exemplo, quando você elimina dependências de operações específicas, pelo isolamento e encapsulamento de cada operação, torna mais fácil a reutilização de uma operação em contextos diferentes. A mesma coisa também pode acontecer quando você remove dependências algorítmicas e de representação.

Os padrões de projeto também tornam uma aplicação mais fácil de ser mantida quando são usados para limitar dependências de plataforma e dar uma estrutura de camadas a um sistema. Eles melhoram a facilidade de extensão ao mostrar como estender hierarquias de classes e explorar a composição de objetos. O acoplamento reduzido também melhora a facilidade de extensão. Estender uma classe isoladamente é mais fácil se a classe não depender de muitas outras classes. (GAMMA et al., 2000, p. 37-40).

2.12 *FRAMEWORKS* (ARCABOUÇOS DE CLASSES)

Um ***framework*** é um conjunto de classes cooperantes que constroem um projeto reutilizável para uma determinada categoria de *software*. Por exemplo, um *framework* pode ser orientado à construção de editores gráficos para diferentes domínios, tais como desenho artístico, composição musical e sistemas de CAD para mecânica. Um outro *framework* pode lhe ajudar a construir compiladores para diferentes linguagens de programação e diferentes processadores. Um outro, ainda, pode ajudar a construir aplicações para modelagem financeira. Você customiza um *framework* para uma aplicação específica através da criação de subclasses específicas para a aplicação, derivadas das classes abstratas do *framework*.

O *framework* dita a arquitetura da sua aplicação. Ele irá definir a estrutura geral, sua divisão em classes e objetos e em consequência as responsabilidades-chave das classes de objetos, como estas colaboram, e o fluxo de controle. Um *framework* predefine esses parâmetros de projeto, de maneira que você, projetista/implementador da aplicação, possa se concentrar nos aspectos específicos da sua aplicação. Um *framework* captura as decisões de projeto que são comuns ao seu domínio de aplicação.

Assim, *frameworks* enfatizam reutilização de projetos em relação à reutilização de código, embora um *framework*, geralmente, inclua subclasses concretas que você pode utilizar diretamente. A reutilização neste nível leva a uma inversão de controle entre a aplicação e o *software* sobre a qual ela está baseada. Quando você usa um *toolkit* (ou, pelos mesmos motivos, uma biblioteca convencional de sub-rotinas) escreve o corpo principal da aplicação e chama o código que quer reutilizar. Quando usa um *framework*, você reutiliza o corpo principal e escreve o código que *este* chama.

Você terá que escrever operações com nomes e convenções de chamada já especificadas; porém isso reduz as decisões de projeto que você tem que tomar. Como resultado, você pode não somente construir aplicações mais rapidamente, como também construí-las com estruturas similares. Elas são mais fáceis de manter e parecem mais consistentes para seus usuários. Por outro lado, você perde alguma liberdade criativa, uma vez que muitas decisões de projeto já terão sido tomadas por você.

Se as aplicações são difíceis de projetar, e os *toolkits* são ainda mais difíceis, os *frameworks*, então, são os mais difíceis entre todos. O projetista de um *framework* aposta que uma arquitetura funcionará para todas as aplicações do domínio. Qualquer mudança substancial no projeto do *framework* reduziria seus benefícios consideravelmente, uma vez que a principal contribuição de um *framework* para uma aplicação é a arquitetura

que ele define. Portanto, é imperativo projetar o *framework* de maneira que ele seja tão flexível e extensível quanto possível.

Além disso, porque as aplicações são tão dependentes do *framework* para o seu projeto, elas são particularmente sensíveis a mudanças na interface do *framework*. À medida que um *framework* evolui, as aplicações têm que evoluir com ele. Isso torna o acoplamento fraco ainda mais importante; de outra maneira, mesmo pequenas mudanças no *framework* teriam grandes repercussões.

Os tópicos de projeto que acabamos de discutir são os mais críticos para o projeto de um *framework*. Um *framework* que os trata através do uso de padrões de projeto tem muito maior probabilidade de atingir altos níveis de reusabilidade de projeto e código, comparado com um que não usa padrões de projeto. *Frameworks* maduros comumente incorporam vários padrões de projeto. Os padrões ajudam a tornar a arquitetura do *framework* adequada a muitas aplicações diferentes, sem necessidade de reformulação.

Um benefício adicional é obtido quando o *framework* é documentado com os padrões de projeto que ele usa. Pessoas que conhecem os padrões obtêm rapidamente uma compreensão do *framework*. Mesmo pessoas que não os conhecem podem se beneficiar da estrutura que eles emprestam à sua documentação. A melhoria da documentação é importante para todos os tipos de *software*, mas é particularmente importante para *frameworks*. Os *frameworks* têm uma curva de aprendizado acentuada, que tem que ser percorrida antes que eles se tornem úteis.

Embora os padrões de projeto não possam achatar a curva de aprendizado completamente, podem torná-la mais suave, ao fazer com que os elementos-chave do projeto do *framework* se tornem mais explícitos.

Como padrões e *frameworks* têm algumas similaridades, as pessoas frequentemente se perguntam em que esses conceitos diferem. Eles são diferentes em três aspectos principais.

1. **Padrões de projeto são mais abstratos que *frameworks***. Os *frameworks* podem ser materializados em código, mas somente exemplos de padrões podem ser materializados em código. Um ponto forte dos *frameworks* é que podem ser escritos em uma linguagem de programação, sendo não apenas estudados, mas executados e reutilizados diretamente. Em contraposição, os padrões de projeto deste livro têm que ser implementados cada vez que eles são usados. Os padrões de projeto também explicam as intenções, custos e benefícios (*trade-offs*) e consequências de um projeto.

2. **Padrões de projeto são elementos de arquitetura menores que *frameworks***. Um *framework* típico contém vários padrões de projeto, mas a recíproca nunca é verdadeira.

3. **Padrões de projeto são menos especializados que *frameworks***. Os *frameworks* sempre têm um particular domínio de aplicação. Um *framework* para um editor gráfico poderia ser usado na simulação de uma fábrica, mas ele não seria confundido com um *framework* para simulação. Em contraste, os padrões de projeto, neste catálogo, podem ser usados em quase qualquer tipo de aplicação. Embora padrões de projeto mais especializados que os nossos sejam possíveis (digamos, padrões para sistemas distribuídos ou programação concorrente), mesmo estes não ditariam a arquitetura de uma aplicação da maneira como um *framework* o faz.

Os *frameworks* estão se tornando cada vez mais comuns e importantes. Eles são a maneira pela qual os sistemas orientados a objetos conseguem a maior reutilização. Aplicações orientadas a objetos maiores terminarão por consistir-se de camadas de *frameworks* que cooperam uns com os outros. A maior parte do projeto e do código da aplicação virá dos *frameworks* que ela utiliza ou será influenciada por eles. (GAMMA et al., 2000, p. 36-38).

3 APLICABILIDADE DOS PADRÕES NOS MODELOS DE PROCESSO DE ENGENHARIA DE *SOFTWARE*

Além dos padrões de projeto já apresentados, outros padrões foram identificados e são bastante utilizados:

- **Padrões de Análise**: propõem soluções para a etapa de análise de sistemas.
- **Padrões de Interface**: propõem soluções para as interfaces dos aplicativos.
- **Padrões de Processo**: apresentam soluções para as etapas de desenvolvimento, gerenciamento de configurações e testes.
- **Padrões Organizacionais**: descrevem soluções já aferidas para organizar e gerenciar pessoas envolvidas no processo de desenvolvimento.

Padrões de análise podem ser aplicados na fase de levantamento e documentação dos requisitos, quando se usa o modelo Cascata ou Incremental; na etapa da Prototipação; no *workflow* de análise e projeto no RUP; na etapa das interações no modelo de processo do XP. Estes padrões auxiliam na especificação de requisitos de domínio, por exemplo: locar, vender, manter recursos.

Na etapa de elaboração do projeto, os padrões arquiteturais apresentam soluções para a adequada escolha da estrutura do aplicativo. Isso pode ser visualizado durante as etapas de projeto do modelo Cascata, na atividade de engenharia do modelo Espiral, na etapa de projeto do modelo Incremental etc.

De forma geral, o padrão de arquitetura vai definir quais serão os componentes utilizados e a forma de relacionamento e comunicação utilizada por eles. Já os padrões de projeto são usados para fazer o refinamento destes componentes e sua forma de comunicação.

Outras aplicações dos padrões:

- No projeto e otimização de banco de dados relacional no intuito de otimizá-los.
- Aplicação de padrões de processo e organizacionais no processo de desenvolvimento de *software* interativo, em todas as etapas do projeto.
- Nos testes, que podem ser utilizados pelos engenheiros de *software* na criação de casos de testes, apresentando quais das técnicas de testes poderiam ser aplicadas para um determinado problema.
- Processos de reengenharia de *software*.
- No **IHC** (Interação Homem Computador), nas seguintes situações:

- **Metas de Negócio**: descrevem o objetivo do sistema;
- **Padrões de Postura**: descrevem a estrutura que é comumente usada por sistemas relacionados;
- **Padrões de Experiência**: descrevem as atividades do usuário;
- **Padrões de Tarefas**: propõem soluções para problemas pequenos (do cotidiano) do usuário;
- **Padrões de Ação**: especificam o uso de *widgets* bem conhecidos ou descrevem *widgets* customizáveis.

Widget é um componente de uma interface gráfica do usuário, o que inclui janelas, botões, menus, ícones, barras de rolagem etc.
FONTE: Disponível em: <https://pt.wikipedia.org/wiki/Widget>. Acesso em: 6 abr. 2016.

Os padrões de IHC são divididos em duas categorias:

- **Padrões de IHC**: envolvem tudo o que está relacionado com os usuários do sistema.

- **Padrões de Interface com o Usuário**: abordam tudo o que se relaciona com a interface do sistema.

Os padrões IHC detalham as tarefas que são realizadas pelo usuário no sistema, e como o sistema as gerencia. O resultado final apresentado pelo uso de padrões é a especificação da interação e da interface, com o usuário final. Logo, entende-se que os padrões podem ser utilizados como uma forma de comunicação que cria um vocabulário único e padronizado entre a equipe de desenvolvimento e o usuário dos aplicativos.

Padrões IHC também podem ser utilizados para planejar e aplicar todos os tipos de avaliações durante as etapas do projeto de desenvolvimento de *software*.

RESUMO DO TÓPICO 2

Neste tópico você aprendeu que:

- Os padrões são ferramentas importantes para a análise e o projeto de uma solução de *software*. Oferecem ao arquiteto a possibilidade de conceitualizar uma solução em diferentes níveis. São, ainda, uma forma valiosa de comunicar conceitos entre os profissionais de *software*. (HOFSTADER, 2007).

- Arquiteturas e projetos de solução devem mapear os requisitos funcionais. Ao implementar padrões no projeto de uma solução, o arquiteto pode criar estruturas específicas de domínio para fazer com que a solução de *software* seja totalmente consistente. Considerando que a arquitetura e o projeto de sistemas são altamente subjetivos, a melhor forma de avaliar uma estrutura é saber se cumpre os requisitos funcionais definidos e se é flexível, extensível e permite manutenção. (HOFSTADER, 2007).

- Não é preciso ter um processo formal para definir uma solução baseada em padrões. Uma abordagem ágil para codificar os requisitos funcionais e refratária a estruturas baseadas em padrões poderá ser ideal se o domínio não estiver inicialmente bem definido. Ao contrário, se o domínio estiver bem definido, o projeto da solução poderá ser iniciado com padrões e modificado para cumprir outros requisitos, como desempenho. (HOFSTADER, 2007).

AUTOATIVIDADE

1 Quando devemos usar o padrão Singleton?

2 Forneça (descreva) um exemplo para o uso do padrão Singleton.

TÓPICO 3

PROCESSO DE TOMADA DE DECISÃO

1 INTRODUÇÃO

De certa forma, todas as empresas vivem de projetos. Mesmo aquelas cujo resultado final do trabalho produzido não tenha sua origem em projetos específicos. Logo, a tomada de decisão em projetos é uma área com crescimento exponencial.

Tomada de decisão e gestão de projetos são conceitos "casados". A gestão de projetos auxilia estrategicamente a tomada de decisões nas organizações, tornando-as mais competitivas, pois permite o refinamento de habilidades e competências acerca de todos os envolvidos e afetados por estas decisões.

No desenvolvimento de *software* as decisões são volumosas e corriqueiras. Decide-se sobre a melhor tecnologia, a matriz de responsabilidade do projeto, o padrão de construção, a forma de comunicação, as aquisições. Muitas são as decisões que devem ser tomadas diariamente para garantir o sucesso do projeto.

A seguir, apresentamos um levantamento teórico acerca do tema de tomada de decisões.

2 TOMADA DE DECISÃO EM PROJETOS DE *SOFTWARE*

No que se refere à natureza dos problemas e do projeto no desenvolvimento de *software*, as decisões podem envolver várias pessoas; podem apresentar distintos níveis de risco e complexidade; e necessitar de diferentes ações por parte dos decisores para garantir o sucesso daquilo que será executado/construído. Geralmente, o ato de decidir está associado à figura dos gerentes, porém, todos os colaboradores, de forma geral, tomam decisões diariamente, pois precisam resolver problemas diversos. No caso de desenvolvimento de *software*, isso ocorre desde as reuniões iniciais do projeto até a entrega final.

Durante o desenvolvimento de um *software*, é comum ocorrer a famosa "alteração de escopo", e com isso é necessário adotar critérios para a tomada de decisão, a fim de garantir o menor impacto em custo, prazo e qualidade.

Mas, as mudanças, podem e devem ser administradas. Para que isso aconteça, devem primeiro ser entendidas e em seguida, documentadas para análise. Uma documentação adequada da alteração a ser realizada, é garantia de um correto levantamento de prazos, custos e gestão da qualidade. Como efeito colateral benéfico desta prática, podemos citar a minimização dos riscos do projeto, que poderiam ser causados pela situação de mudança de escopo.

Uma alteração de escopo em projetos de *software* desencadeia uma tomada de decisões em cascata através da: realocação de pessoas; contratação de novas pessoas; aquisição de novas tecnologias; definição de um novo cronograma, e assim por diante. Logo, o gerente de projetos é solicitado a coordenar e integrar atividades de múltiplas linhas funcionais.

Assim como o desenvolvimento de *software*, a decisão envolve processos e acontece dentro de fases ou etapas que ocorrem sequencialmente. No desenvolvimento de *software* sempre há mais de um caminho a seguir, quando temos que decidir. Sempre há mais de uma alternativa de solução para um algoritmo; sempre há mais de uma tecnologia disponível para dar suporte na arquitetura de solução; sempre há mais de uma linguagem de programação; sempre há mais de uma forma de escrever a codificação do programa.

Todos os envolvidos no processo de desenvolvimento de *software*, são tomadores de decisão e se deparam com algumas restrições quando estão diante de uma decisão importante:

- O tempo e a corretude da decisão estão diretamente relacionados pela rapidez do seu processo mental, lógica e aritmética;
- Seus valores internos;
- Seu nível de conhecimento, que será usado tanto na interpretação do problema, quanto ao domínio dos recursos disponíveis para uso na elaboração e da solução.

Outros fatores também podem afetar a tomada de decisão em projetos: o estresse, a pressão do tempo, o envolvimento com a tarefa, a cultura, o nível social, o sexo, a religião, os costumes e as crenças, a ética moral e a ética profissional, a saúde física e a mental, a influência familiar, e o fator emocional na hora exata da tomada de decisão.

3 MODELOS DO PROCESSO DECISÓRIO

As decisões são amparadas em modelos. Estes, apresentam-se da seguinte forma:

- **Modelo Clássico ou Burocrático**: parte do princípio de que o decisor analisa o possível resultado para depois definir os meios para alcançá-lo.

- **Modelo Comportamentalista**: considera o comportamento das pessoas no contexto das decisões.

- **Modelo Normativo**: tem como preocupação central aquilo que deverá ser feito.

- **Modelo Racional**: seu principal objetivo é potencializar os objetivos da organização.

Os modelos se relacionam com o contexto de tomada de decisão através da estrutura, cultura, ambiente e clima organizacional. Relacionado a estes fatores podemos citar ambientes de trabalho complexos e dinâmicos; o nível de burocracia das organizações e sua distribuição de poder; a cultura estabelecida na organização em relação às pessoas e suas atividades; as características do decisor (proativo, intuitivo, inovador etc.); e por último, os problemas/assuntos que são o centro das decisões; se eles são urgentes, restritos, bem como seu nível de complexidade e urgência.

Por isso, podemos concluir que a tomada de decisão sofre forte influência pelas características do ambiente e pelas pessoas inseridas no contexto da situação-problema.

3.1 TIPOS, MODELOS E NÍVEIS DE TOMADA DE DECISÃO

Bons tomadores de decisão consideram a complexidade e a natureza dinâmica do mundo dos negócios. Podemos adotar duas formas de pensamentos distintos na tomada de decisão: pensamento linear e pensamento sistêmico.

O pensamento linear trata o problema como sendo único, com solução única e impacto limitado à área do problema em si, sem afetar outros setores organizacionais. Embora prática e simplista, é necessária uma visão mais abrangente em relação ao cenário em análise, e principalmente, sobre o impacto da decisão em todos os âmbitos organizacionais. Este tipo de pensamento ignora as rápidas mudanças sofridas pelo ambiente e pelas pessoas.

Por outro lado, existe o pensamento sistêmico, que é uma forma mais moderna e atual de se pensar nos problemas. Nesta abordagem, os problemas são percebidos de forma globalizada e analisam o impacto do mesmo e da decisão de

solução, em toda a organização. Este pensamento não vê o problema como único, mas relacionado com tudo o que acontece no ambiente.

Neste sentido, o tomador de decisão que se apoia no pensamento sistêmico deve considerar o relacionamento dos sistemas e processos organizacionais, antes de decidir. E, após decidir, deverá analisar o impacto e efeitos da decisão na organização, no intuito de sempre melhorar o processo de tomada de decisão.

3.2 NÍVEIS DE TOMADAS DE DECISÃO EM UMA ORGANIZAÇÃO

Decisões organizacionais ocorrem a todo momento e em todos os setores organizacionais. São três os níveis de tomada de decisão dentro de uma organização:

- **Nível estratégico**: estas decisões norteiam os objetivos organizacionais.

- **Nível tático**: concentram-se num nível abaixo das decisões estratégicas, e geralmente são tomadas pelos gerentes de nível médio. Concentram-se em colocar em prática as ações deliberadas e impostas pelo nível estratégico. São decisões com foco em ações, como por exemplo: determinação de compra, redução de custos etc.

- **Nível operacional**: é o nível inferior da estrutura organizacional e se referem às operações executadas diariamente que cumprem as determinações do nível médio ou tático.

FONTE: Disponível em: <http://www.batebyte.pr.gov.br/modules/conteudo>. Acesso em: 16 mar. 2016.

3.3 NOVOS MODELOS DO PROCESSO DECISÓRIO

O processo decisório compreende a aplicação de diferentes modelos de tomada de decisão, cada um deles pertinente a uma determinada situação. Entre eles, podemos destacar como principais, os modelos racional, processual, anárquico e político.

3.3.1 Modelo racional

O modelo de tomada de decisão racional é o mais sistematizado e estruturado entre todos, pois pressupõe regras e procedimentos pré-definidos, que devem ser seguidos para que se possa atingir um bom resultado. Neste modelo prevalece uma estrutura organizacional burocrática e com regras explicitamente formalizadas.

Neste modelo, os objetivos são alcançados através da resolução de problemas resolvidos de forma procedimental e racional.

3.3.2 Modelo processual

Desmembra e descreve as fases que dão suporte para as atividades decisórias.

Seu fundamento baseia-se em alguns questionamentos: quais são as organizações que atuam nesse tipo de circunstância?; quais são as rotinas e procedimentos utilizados usualmente?; quais são as informações disponíveis?; e quais são os procedimentos padrões utilizados nesses casos? Este modelo se concentra nas fases, atividades e comportamentos do ato de decidir.

O modelo processual é definido da seguinte forma: é amparado em três fases decisórias principais, três rotinas de apoio às decisões e seis grupos de fatores dinâmicos.

As três principais fases decisórias são:

- **Identificação**: identifica a necessidade para a tomada de decisão, analisando o problema e todo o seu contexto.
- **Desenvolvimento**: apresenta uma ou mais soluções para o mesmo problema e analisa o impacto de cada solução proposta.
- **Seleção**: avalia as opções de solução para escolher especificamente uma.

As três rotinas de apoio no modelo processual são:

- **Rotinas de controle**: planejam e determinam o limite da decisão.
- **Rotinas de comunicação**: disseminam a informação da solução para todos os interessados, através dos canais de comunicação disponíveis.
- **Rotinas políticas**: são situações de barganha ou cooptação para a solução dos problemas.

Os seis grupos de fatores dinâmicos são:

- **Interrupções**: remetem às questões internas e externas do ambiente no qual o problema está inserido.
- **Adiantamento de prazos**: reduz o andamento das atividades do processo decisório.
- ***Feedback***: externalização dos resultados das ações na solução dos problemas.
- **Ciclos de compreensão**: envolve procedimentos para melhor compreensão dos problemas e suas respectivas possibilidades de solução.
- **Ciclos de fracasso**: análises que são feitas quando não é possível decidir.

3.3.3 Modelo anárquico

Choo (2003, p. 295) explica que: "[...] o modelo anárquico de decisão pode ser comparado a uma lata de lixo, onde vários tipos de problemas e soluções são atirados pelos indivíduos, à medida que são gerados. A decisão ocorre quando problemas e soluções coincidem".

No modelo anárquico, existem três formas para decidir:

> • Resolução: é a tomada de decisão que ocorre depois de se pensar sobre o problema, por determinado tempo;
> • Inadvertência: uma escolha é adotada rápida e incidentalmente, para outras escolhas serem feitas;
> • Fuga: ocorre quando os problemas abandonam a escolha, quando não há resolução do problema (CHOO, 2003, p. 297).

3.3.4 Modelo político

Este modelo apoia-se em aspectos políticos no processo decisório. Para isso, considera os envolvidos, suas posições e graus de influência. Desse modo, a decisão não é racional, mas sim, baseada na influência e poder de cada ator envolvido.

Questões que norteiam o problema: quais são os canais usados para produzir ações que resolvam um tipo de problema?; quem são os atores e quais suas posições?; e quais são as pressões que estão influenciando?

Este modelo expõe cenários de disputas internas em relação ao poder, *status* e posição hierárquica, que se torna mais acentuado no funcionalismo público que é amplamente influenciado politicamente.

Independente do modelo adotado, o decisor deve sempre se questionar sobre a importância da decisão. Num segundo momento deve estar atento para as consequências da escolha ou do descarte de uma decisão específica.

Cabe lembrar que a decisão é um conjunto de análises, comparações e possibilidades, e que não há como separar tendências e preferências pessoais ao escolher uma opção entre tantas que podem ser assertivas. Mas, lembre-se: a análise pessoal é necessária, uma vez que nem sempre temos total domínio do cenário no qual se faz necessário uma tomada de decisão rápida e eficiente.

3.4 OS TIPOS DE DECISÕES

> Qualquer escolha que se faça numa organização implica numa tomada de decisão. Em todos os níveis e subunidades, as pessoas decidem e assim determinam a quantidade de criação de valor. A tomada de decisão é o processo de responder a um problema, procurando e selecionando uma solução ou ação que irá criar valor para os acionistas da organização, sendo o problema de diversas naturezas, como o de procurar os melhores recursos, decidir como fornecer um serviço ou saber como lidar com um competidor agressivo.

FONTE: Disponível em: <http://www.batebyte.pr.gov.br/modules/conteudo>. Acesso em: 16 mar. 2016.

É prática organizacional a busca por meios que ajudem seus gestores a tomar a melhor decisão dentre tantas opções. Os tipos de decisões facilitam os processos decisórios.

Basicamente, existem dois tipos de decisões:

- **Decisões programadas**: remetem a problemas e situações que são bem entendidas; seu universo é bem delimitado e estruturado; os problemas e soluções se repetem com certa frequência. Por isso, existe muita semelhança nas decisões tomadas.

- **Decisões não programadas**: são aquelas que tratam de problemas atípicos, que não fazem parte da rotina organizacional; os problemas não ocorrem com frequência, por isso, pode não haver histórico de decisões semelhantes para amparar uma decisão que deve ser tomada de forma imediata.

Pelo fato de as decisões não programadas serem tão importantes para as empresas e tão comuns para a gerência, a eficácia de um gerente muitas vezes será julgada de acordo com a qualidade de sua tomada de decisão. Muitas empresas criaram programas de treinamento para ajudar os gerentes na tomada de decisão, pois eles tomam muitas decisões não programadas.

Os gerentes precisam ser estimulados a desaprender ideias antigas e testarem suas habilidades de tomada de decisão. Algumas ideias para promoverem essa melhoria são: coletar novas informações para avaliarem novas alternativas (pesquisas mostram que, na prática, essas ideias não são bem aceitas); converter eventos em oportunidades de aprendizado e assim motivarem-se a encontrar novas respostas e formas de visão para algumas situações; e gerar novas alternativas de comportamento.

FONTE: Disponível em: <http://www.batebyte.pr.gov.br/modules/conteudo>. Acesso em: 16 mar. 2016.

LEITURA COMPLEMENTAR

PADRÕES DE PROJETO DE *SOFTWARE* BASEADO EM COMPONENTES APLICADOS A UM SISTEMA JAVA PARA ACADEMIA DE MUSCULAÇÃO

Jean Wagner Kleemann

Entre diversas metodologias e práticas de desenvolvimento de *software* existentes no mercado atual, muitas soluções e padrões de projeto não proporcionam o desenvolvimento rápido e para sistemas complexos. Entretanto as componentização de *software* juntamente com bons padrões de projeto proporcionam uma solução capaz de gerenciar problemas complexos ligados ao reaproveitamento, não somente de código, mas, independentemente de tecnologias ou plataformas, reduzindo tempo e recursos financeiros. Este trabalho sugere o uso de padrões de projetos relacionados com a engenharia de componente para a criação de um sistema de academia de musculação, destacando as vantagens de tais metodologias e seu relacionamento com a capacidade do sistema crescer e evoluir com facilidade, através do uso destes processos.

1 INTRODUÇÃO

No ambiente de desenvolvimento de *software*, uma questão muito discutida é a produtividade. Os sistemas estão a cada dia mais complexos e muitos padrões de desenvolvimento ou gerenciamento de processos não atendem às expectativas da forma desejada. Com o desenvolvimento baseado em componentes, surge o conceito de "crie uma vez, use onde quiser". Desse modo é possível montar sistemas utilizando componentes já implementados, não sendo necessário escrever uma nova aplicação codificando linha após linha.

Para a implementação de componentes de *software* é necessário o forte uso de padrões em todo processo de desenvolvimento. Consequentemente há uma série de vantagens ao utilizar tais padrões, mencionado por Alur (2002) como sendo reutilizáveis, pois fornecem uma solução pronta que pode ser adaptada para diferentes problemas quando necessário. Os padrões refletem a experiência e conhecimento dos desenvolvedores que utilizaram estes padrões com sucesso em seu trabalho, formado um vocabulário comum para expressar grandes soluções sucintamente.

Além de reduzir o tempo de aprendizado de uma determinada biblioteca de classes, também diminuem o retrabalho, pois quanto mais cedo são usados, menor será o retrabalho em etapas mais avançadas do projeto.

Este trabalho apresenta uma abordagem sobre alguns padrões de projetos associados ao conceito de *software* componentizado, que são capazes de proporcionar o desenvolvimento de *software* de forma rápida, resultando em aplicações capazes de crescerem e evoluírem com facilidade. Para aplicação destes padrões será utilizado o ambiente de uma academia de musculação, onde o resultado final será um sistema que deverá ser utilizado tanto no ambiente *web* como *desktop* e com diversos tipos de persistência de dados.

2 ENGENHARIA DE COMPONENTES E PADRÕES DE PROJETO

Segundo Friedrich Ludwig Bauer (1969, p. 231), "Engenharia de *Software* é a criação e a utilização de sólidos princípios de engenharia a fim de obter *software* de maneira econômica, que seja confiável e que trabalhe eficientemente em máquinas reais".

Outra definição para o conceito de engenharia de *software* é apresentada por Sommerville (2007) como uma disciplina de engenharia relacionada com todos os aspectos de produção de *software*, desde os estágios iniciais de especificação do sistema até a sua manutenção, ou seja, mesmo depois que este entrar em operação. Isso revela a importância da engenharia de *software* dentro de todo o projeto e sua forte ligação com o resultado final do produto.

Já a engenharia de *software* baseada em componentes (*component-based software engineering*, CBSE) é um processo que enfatiza o projeto e a construção de sistemas baseados em computador usando "componentes" de *software* reutilizáveis (PRESMMAN, 2002). A Engenharia de componentes é uma derivação da engenharia de *software*, focada na decomposição dos sistemas, em componentes funcionais e lógicos com interfaces bem definidas, usadas para comunicação entre os próprios componentes, que estão em um nível mais elevado de abstração do que os objetos, com isso a comunicação se dá por meio de mensagens.

A prática da tecnologia de *software* baseada em componentes, baseia-se no desenvolvimento através de componentes reutilizáveis, levando à redução de tempo de desenvolvimento, e facilitando as mudanças e a implantação de novas funcionalidades. Dessa forma, o processo de engenharia de *software* baseada em componentes tem mudado o modo pelo qual sistemas são desenvolvidos, pois desloca a ênfase da programação do *software* para a composição de sistema de *software* com base em componentes (PRESMMAN, 2002).

A engenharia baseada em componentes possui as etapas de coleta de requisitos e passa por um projeto arquitetural, da mesma forma que a engenharia se *software* tradicional. Porém, a partir desse ponto começa a se diferenciar, pois começam a ser analisados os requisitos com o objetivo de buscar módulos que

sejam mais adequados à composição, ao invés de iniciar a construção e partir para tarefas de projeto mais detalhadas.

Ao fazer essa análise dos subconjuntos ou módulos do sistema, pode-se fazer o uso de componentes já existentes, sendo componentes próprios ou comerciais. Segundo Desmond D'Sousa (1998) um componente é um pacote coerente de artefatos de *software* que pode ser desenvolvido independentemente e entregue como unidade e que pode ser composto, sem mudança, com outros componentes para construir algo maior.

O Reuso é o objetivo principal da engenharia de *software* baseada em componentes. Não se trata somente de reutilização de código, pois abrange também os artefatos envolvidos durante todas as etapas de desenvolvimento. Com isso os riscos são menores ao usar um componente já existente em relação ao desenvolvimento de algo a partir do zero. Também ocorre o aumento da produtividade, tendo em vista a redução de esforços pela equipe de desenvolvimento. Seguindo a ideia "crie uma vez, use onde quiser". Por sua vez, a qualidade e confiabilidade do produto são maiores, pois o componente reutilizado já foi testado e aprovado.

Os Padrões de projeto são um grupo de práticas para definir o problema, a solução e em qual situação aplicar esta solução e suas consequências no desenvolvimento de *software*. Os padrões são desenvolvidos a partir de experiências práticas em projetos reais. É resultado da experiência de arquitetos com experiência, que aplicam estes padrões e experimentam o resultado em seus próprios problemas. A experiência desses arquitetos certamente proporciona a abordagem correta para um problema específico, da forma mais indicada.

Fazendo uso de tais padrões, temos como resultado uma qualidade de código, permitindo a reutilização de forma flexível e expansível. Por se tratarem de questões que nasceram de um problema real, e a solução foi testada e documentada, os padrões aumentam de forma significativa a produtividade e qualidade, pois a solução já está definida.

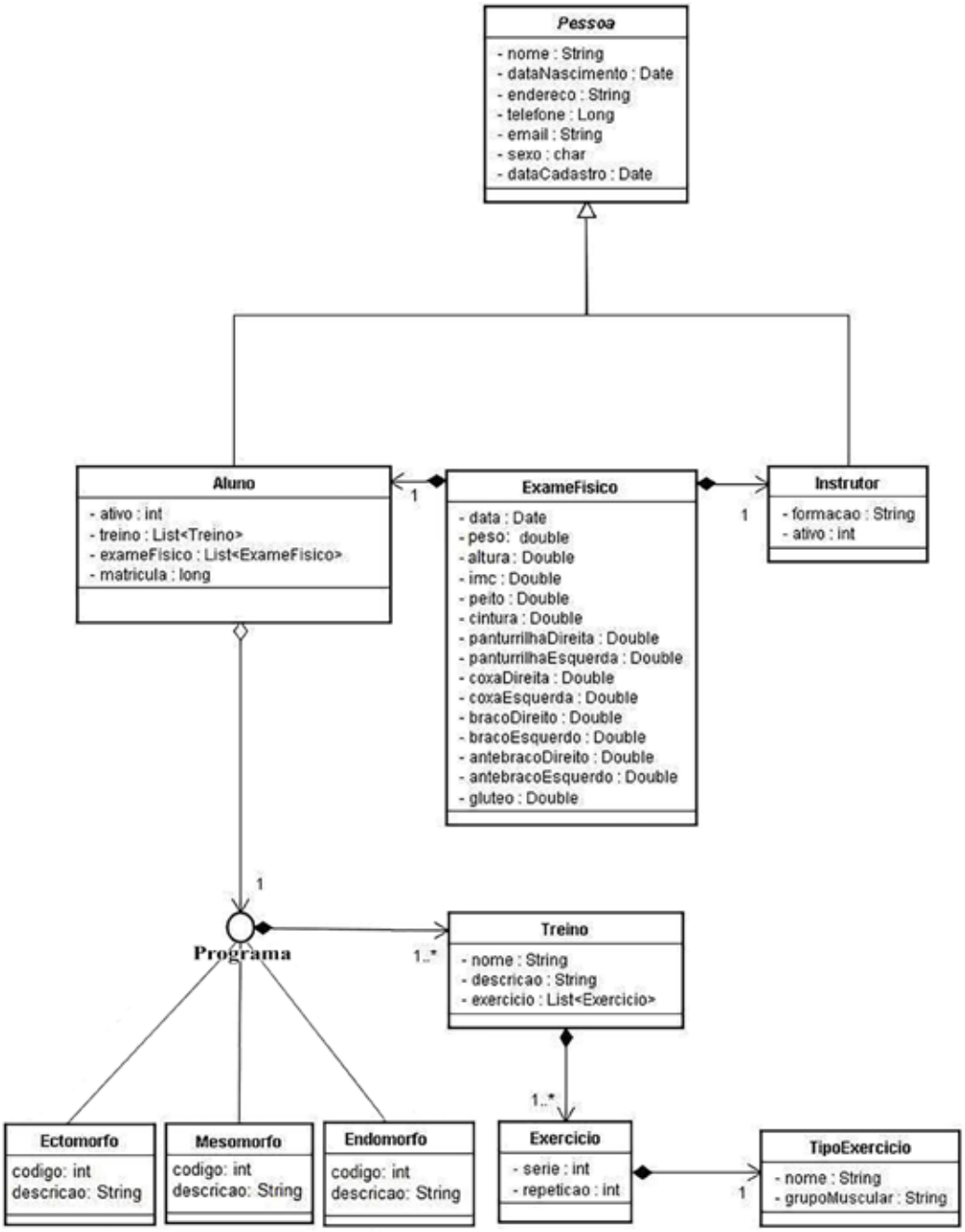
3 DESIGN DO SISTEMA

A proposta de sistema que será apresentada consiste em gerenciar um ambiente de academia de musculação, sendo de responsabilidade do sistema distribuir rotinas de exercícios conforme o tipo físico de cada aluno, que será determinado após um exame físico. Dessa forma não será necessário gerenciar treinos individuais para cada aluno da academia, pois todo aluno sempre será categorizado em um dos três tipos físicos existentes, sendo eles: ectomorfo (abaixo do peso), mesomorfo (peso ideal) e endomorfo (acima do peso).

Este sistema deverá ser independente de sistema operacional e poderá operar tanto em um ambiente *desktop* quanto *web*. Para isso é sugerida a utilização da tecnologia Java, que além de suprir estes requisitos com facilidade, é uma tecnologia muito relevante no mercado atual. O sistema também deve permitir formas diferentes de persistência dos dados definidos em tempo de execução, podendo variar entre JPA, JDBC, EJB ou até mesmo persistência em memória. Através desses requisitos será abordada uma estrutura capaz de solucionar tais solicitações, fazendo o uso de alguns padrões de arquitetura, comportamentais, estruturais e criacionais.

A seguir será representado um diagrama de classe com a proposta de modelagem para este ambiente. Através do diagrama representado a seguir pela Figura 1, consta uma entidade chamada pessoa, que foi criada após verificar semelhanças entre atributos de aluno e instrutor. Dessa forma a classe pode ser estendida pelo conceito de herança por demais entidades que caracterizam uma pessoa, que porventura venham a ser criadas, como por exemplo: funcionário ou usuário. Assim será aplicado o reaproveitamento da estrutura já existente.

O exame físico necessita de um aluno e um instrutor, e nele constam as medidas do aluno no momento do exame. Ao inserir um exame no sistema, automaticamente será analisado qual o tipo físico da pessoa, através de um cálculo simples de IMC (Índice de Massa Corporal). Com base neste cálculo o aluno receberá um programa específico para seu perfil em tempo de execução, através do conceito de interface.



FONTE: Disponível em: <http://www.linhadecodigo.com.br/artigo/3130/padroes-de-projeto-de-software-baseado-em-componentes-aplicados-a-um-sistema-java-para-academia-de-musculacao.aspx#ixzz42VWgd3ws>. Acesso em: 6 abr. 2016.

RESUMO DO TÓPICO 3

Neste tópico você aprendeu que:

- A tomada de decisão é indispensável e crucial para a administração das organizações.
- O processo decisório é complexo e contém várias etapas.
- O pensamento linear enfatiza que os problemas têm apenas uma solução, não afetam o restante da organização e, uma vez descoberta a solução, esta permanecerá constantemente válida.
- O pensamento sistêmico afirma que os problemas são complexos, têm mais de uma causa e mais de uma solução, e estão inter-relacionados com o restante da organização.
- Diferentes níveis organizacionais tomarão tipos diferentes de decisão. A alta gerência será responsável por determinar as metas estratégicas de uma empresa, ao passo que os gerentes intermediários tomarão decisões táticas ou administrativas.
- O nível organizacional mais baixo da administração, a supervisão, tomará decisões operacionais.
- O sucesso das decisões depende da capacidade analítica dos gerentes.
- A tomada de decisão dentro da organização contemporânea de negócios envolve todos os tipos e estilos de solução de problemas.
- Embora um tipo e estilo em particular possa ser mais eficaz do que outros, em uma situação específica, todas as organizações são confrontadas com uma variedade bastante complexa de desafios que exigem uma gama de estilos de solução de problemas.

AUTOATIVIDADE

Explique o que são decisões não programadas.

REFERÊNCIAS

AMBLER, Scott W. **Análise de projeto orientado a objeto**. 2. ed. Rio de Janeiro: Infobook, 1998.

ANDRADE, C. A. V. **Metodologia de análise de sistemas**. 8° Módulo. ESAB – Escola Superior Aberta do Brasil: Vila Velha, 2007.

BAHIA. Secretaria da Fazenda do Estado da Bahia. **Manual de padrões**: desenvolvimento web e MVC. Versão 01.00.00. Salvador (BA), setembro de 2013. Disponível em: <http://www.sefaz.ba.gov.br/downloads/prt/manuais/padroes/manual_de_padroes_desenvolvimento_web_mvc.doc>. Acesso em: 1 mar. 2016.

BEZERRA, E. **Princípios de análise e projeto de sistemas com UML**. Rio de Janeiro: Elsevier, 2007.

BIEMAN, J. M.; OTT, L. M. Measuring functional cohesion. **IEEE Transactions on Software Engineering**, Nova York, EUA, v. 20, n. 8, p. 308-320, ago. 1994.

BOOCH, G. **Object-oriented analysis and design with applications**. Redwood City, USA: Benjamin, 2006.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. **UML**: guia do usuário. Rio de Janeiro: Campus, 2006.

BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. **UML**: guia do usuário. Rio de Janeiro: Elsevier, 2005.

BORGES, L.; CLINIO, A. L. Programação Orientada a Objetos com C++. Apostila. Disponível em: <http://webserver2.tecgraf.puc-rio.br/ftp_pub/lfm/CppBorgesClinio.pdf>. Acesso em: 6 mar. 2015.

BROOKS, F. P. No silver bullet: essence and accidents of software engineering. **Computer**, New York, v. 20, n. 4, p. 10-19, Apr. 1987.

CHOO, C. W. **A organização do conhecimento**: como as organizações usam a informação para criar significado, construir conhecimento e tomar decisões. São Paulo: Editora Senac São Paulo, 2003.

DEDENE, B.; SNOECK, M. M.E.R.O.DE.: A Model-Driven Entity-Relationship Object-Oriented Development Method. **ACM SIGSOFT, Software Engineering Notes**, v. 19, n. 3, p. 51-61, julho 1994.

DESCHAMPS, F. **Padrões de projeto**: uma introdução. 2009. Disponível em: <http://s2i.das.ufsc.br/tikiwiki/apresentacoes/padroes_de_projeto.pdf>. Acesso em: 15 fev. 2009.

DHAMA, H. Quantitative models of cohesion and coupling in software. **Journal of Systems and Software**, Orlando, EUA, v. 29, n. 4, abr. 1995.

DINSMORE, C.; CAVALIERI, A. **Como se tornar um profissional em gerenciamento de projetos:** livro-base de "Preparação para Certificação PMP - Project Management Professional". Rio de Janeiro: QualityMark, 2003.

FUNCK, M. A. **Estudo e aplicação de métricas da qualidade do processo de desenvolvimento de aplicações em banco de dados.** Trabalho de Conclusão de Curso Ciências da Computação. Universidade Regional de Blumenau, 1994.

GAMMA, E. et al. **Padrões de projeto**: soluções reutilizáveis de software orientado a objetos. Porto Alegre, Brasil: Bookman, 2000.

GRAHAM, I. **Object Oriented Methods**. Addison-Wesley: Wokingham, England, 1994.

GUEDES, G. T. A. **Uml 2**: guia prático. São Paulo: Novatec, 2007.

GUEDES, G. T. A. **Uml 2**: uma abordagem prática. 2. ed. São Paulo: Novatec, 2011.

HAMILTON, K.; MILES, R. **Learning UML 2.0**. O'Reilly, 2006.

HILSDALE, E.; KERSTEN, M. **Aspect-Oriented Programming with AspectJ** (tutorial 24). Proceedings of OOPSLA 2004 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications. Vancouver, Canadá, 2004.

HOFSTADER, Joseph. **Usando padrões para definir uma solução de software**. 12 de fevereiro de 2007. Disponível em: <https://msdn.microsoft.com/pt-br/library/bb190165.aspx>. Acesso em: 2 maio 2016.

HUMPHREY, W. S. **A Discipline for Software Engineering, SEI Series on Software Engineering**, Addison-Wesley Pub Co, 1998.

JACOBSON, I. **Object-Oriented Development in an Industrial Environment**. Anais do OOPSLA 87. 1987.

JACOBSON, I.; BOOCH, G.; RUMBAUGH, J. **The unified software development process**. Reading, Massachusetts, EUA: Addison-Wesley, 1999.

JACOBSON, I.; NG, P. W. **Aspect-oriented software development with use cases**. Reading, Massachusetts, EUA: Addison-Wesley, 2004.

JACOBSON, Ivar et al. **Object oriented software engineering**: a use case driven approach. Wokingham: Addison Wesley, 1992.

KRUCHTEN, P. **Introdução ao RUP - Rational Unified Process**. Rio de Janeiro, Brasil: Ciência Moderna, 2003.

LARMAN, C. **Utilizando UML e padrões**. 3. ed. Porto Alegre: Bookman, 2007.

LORENZ, Mark; KIDD, Jeff. **Object-oriented software metrics**: a practical guide. New Jersey: PTR Prentice Hall, 1994.

MEDEIROS, L. M. **Análise da complexidade de software baseada em objetos**. 2000. Monografia apresentada como requisito básico à obtenção do grau de Bacharel em Ciência da Computação, do Curso de Ciência da Computação do Centro Universitário do Triângulo, Uberlândia, dezembro de 2000.

MELO, A. C. **Desenvolvendo aplicações com UML 2.2**. 3. ed. Rio de Janeiro: Brasport, 2011.

MOLLER, Kurt. H.; PAULISH, Daniel J. **Software metrics**: a practitioneris guide to improved product development. Los Alamitos: IEEE, 1993.

OMG. **Unified Modeling Language - Superstructure specification Versão 2.0**. OBJECT MANAGEMENT GROUP: Nedham, Massachusets, EUA, 2005.

OMG. **Unified Modeling Language**: infrastructure. V2.1.2, 2007.

PIMENTEL, A. R. **Uma abordagem para projeto de software orientado a objetos baseado na teoria de projeto axiomático**. Curitiba, 2007. 193 p.

PIVA, G. D. **Análise e gerenciamento de dados**. Manual de Informática Centro Paula Souza, v. 3, São Paulo, 2010.

PMI. Project Management Institute. **A guide to the project management body of knowledge.** Syba: PMI Publishing Division, 2000. Disponível em: <http://www.pmi.org>. Acesso em: 11 fev. 2015.

PMI. Project Management Institute. **A guide to the project management body of knowledge**. 4. ed. Newton Square: Project Management Institute, 2008.

PMI. Project Management Institute. **Project Management Body of Knowledge - PMBoK**. Pennsylvania: USA, 2004.

PMI. Project Management Institute. **Project Management Body of Knowledge - PMBoK**. 5. ed. Pennsylvania: USA, 2013.

PRESSMAN, R. S. **Software engineering**: a practitioner's approach. 6. ed. Nova York: McGraw-Hill, 2005.

PRESSMAN, R. S. **Software engineering**: a practitioner's approach. 6. ed. Nova York: McGraw-Hill, 1995.

ROSENBERG, L. **Applying and interpreting object oriented metrics**. 2. ed. McGraw-Hill, 1998.

RUMBAUGH, J. E. et al. **Object-oriented modeling and design**. Englewood Cliffs, New Jersey: Prentice Hall, 1991.

RUMBAUGH, J. et al. **Modelagem e projetos baseados em objetos**. Editora Campus: Rio de Janeiro, 1994.

SHEPPERD, Martin. **Foundation of software measurement**. New York: Prentice Hall, 1995.

SILVA, Ricardo Pereira e. **UML 2 em modelagem orientada a objetos**. Florianópolis: Visual Books, 2007.

SOMMERVILLE, I. **Engenharia de software**. 9. ed. São Paulo: Addison Wesley, 2011.

SOMMERVILLE, I. **Software engineering**. 8. ed. Pearson Education, 2008.

SOMMERVILLE, I. **Software engineering**. Wokingham, England: Addison-Wesley, 1989.

TACLA, C. A. **Análise e projeto OO e UML 2.0**. Departamento Acadêmico de Informática. Universidade Tecnológica Federal do Paraná, 2010.

TONSIG, S. L. **Engenharia de software**: análise e projeto de sistemas. 2. ed. Rio de Janeiro: Ciência Moderna, 2008.

TRENTIM, M. H. **Gerenciamento de projetos**. Guia para as certificações CAPM e PMP. 2. ed. Atlas, 2011.

VARGAS, Ricardo Viana. **Gerenciamento de projetos**: estabelecendo diferenciais competitivos. 7. ed. Rio de Janeiro: Brasport, 2009.

VIDEIRA, C. A. E. **UML, metodologias e ferramentas case**. 2. ed. Lisboa: Centro Atlântico, 2008. 2 v.

WINCK, D. V.; GOETTEN JUNIOR, V. **AspectJ**: programação orientada a aspectos com Java. São Paulo: Novatec, 2006.