



MÓDULO DE:

METODOLOGIA de ANÁLISE de SISTEMAS

AUTORIA:

Ms. Carlos Valente

Módulo de: METODOLOGIA de ANÁLISE de SISTEMAS

Autoria: Ms. Carlos Valente

Primeira edição: 2007

Todos os direitos desta edição reservados à
ESAB – ESCOLA SUPERIOR ABERTA DO BRASIL LTDA
<http://www.esab.edu.br>
Av. Santa Leopoldina, nº 840/07
Bairro Itaparica – Vila Velha, ES
CEP: 29102-040
Copyright © 2007, ESAB – Escola Superior Aberta do Brasil

A PRESENTAÇÃO

O processo de desenvolvimento de sistemas necessita de métodos específicos para se obter softwares de qualidade. Através de metodologias adequadas o trabalho do analista se profissionaliza.

O BJETIVO

Definir as principais funções de um analista de sistemas, e as principais ferramentas para desenvolver sistemas de média e grande complexidade. Possibilitar a escolha da melhor metodologia e ferramentas conforme as necessidades dos usuários de sistemas.

E MENTA

Apresentar os fundamentos das metodologias de análise de sistemas consagradas no mercado. Detalhar o processo de modelagem e prototipagem no desenvolvimento de sistemas. Definir as metodologias e as suas principais ferramentas. Descrever os diagramas da UML.

S OBRE O AUTOR

- Professor e Consultor de Tecnologia de Informação

- Doutorando (ITA) e Mestre (IPT) em Engenharia de Computação, Pós-Graduado em Análise de Sistemas (Mackenzie), Administração (Luzwell-SP), e Reengenharia (FGV-SP). Graduado/Licenciado em Matemática.
- Professor e Pesquisador da Universidade Anhembi Morumbi, UNIBAN, e ESAB (Ensino a Distância). Autor de livros em Conectividade Empresarial. Prêmio em E-Learning no Ensino Superior (ABED/Blackboard).
- Consultor de T.I. em grandes empresas como Sebrae, Senac, Granero, Transvalor, etc. Viagens internacionais: EUA, França, Inglaterra, Itália, Portugal, Espanha, etc.

SUMÁRIO:

UNIDADE 1	8
Introdução à Análise de Sistemas	8
UNIDADE 2	12
Revisão Geral de Engenharia de Software	12
UNIDADE 3	15
Estrutura Geral de um Desenvolvimento Incremental e Iterativo	15
UNIDADE 4	19
Processo de Desenvolvimento de Software: Levantamento de Requisitos.....	19
UNIDADE 5	25
Processo de Desenvolvimento de Software: Análise de Requisitos.....	25
UNIDADE 6	29
Processo de Desenvolvimento de Software: Projeto, Implementação, Testes e Implantação	29
UNIDADE 7	33
O que é Metodologia ? O que é Análise de Sistemas ? O contexto dentro de Engenharia de Software	33
UNIDADE 8	37
O que faz um Analista de Sistemas ? A evolução das Metodologias e ferramentas CASE	37
UNIDADE 9	45
Metodologias de Análise de Sistemas: Visão Geral	45
UNIDADE 10	49
Metodologia Estruturada	49
UNIDADE 11	53
DFD – Diagrama de Fluxo de Dados	53
UNIDADE 12	60
MER e DER – Modelo e Diagrama de Entidades e Relacionamentos	60
UNIDADE 13	66
Evolução da Metodologia Estruturada.....	66
UNIDADE 14	70

Modelagem de Sistemas Orientados a Objetos	70
UNIDADE 15	74
O que é Modelo ?.....	74
UNIDADE 16	79
Princípios da Modelagem.....	79
UNIDADE 17	84
UML – Unified Modeling Language	84
UNIDADE 18	90
O Paradigma da Orientação de Objetos	90
UNIDADE 19	95
Encapsulamento, Polimorfismo e Herança	95
UNIDADE 20	99
Linguagens Orientada a Objetos.....	99
UNIDADE 21	102
Diagrama de Casos de Uso	102
UNIDADE 22	105
Casos de Uso: Formato, Detalhamento e Abstração	105
UNIDADE 23	109
Caso de Uso – Atores e Relacionamentos.....	109
UNIDADE 24	113
Visão Geral dos Diagramas da UML e as suas categorias	113
UNIDADE 25	117
Diagrama de Classes e Diagrama de Estrutura	117
UNIDADE 26	119
Diagrama de Componentes e Diagrama de Instalação.....	119
UNIDADE 27	122
Diagrama de Objetos e Diagrama de Pacotes	122
UNIDADE 28	125
Exemplo prático: apresentação geral	125
UNIDADE 29	127
Exemplo prático: Diagrama de Classes	127
UNIDADE 30	130

Exemplo prático: Mecanismos e Componentes	130
GLOSSÁRIO	134
REFERÊNCIAS	151

UNIDADE 1

Introdução à Análise de Sistemas

Objetivo: Contextualizar historicamente o início da Análise de Sistemas.

ANÁLISE DE SISTEMAS: COMO SURGIU?

A ideia de sistema surgiu após a Primeira Guerra Mundial, como resultado do crescimento das organizações modernas e da necessidade de seu controle. E com a evolução natural das indústrias possibilitou a produção dos computadores.

A definição de Sistema sugerida pelo American National Standards Institute (ANSI) é:

Sistema, para processamento de dados, é o conjunto de pessoas, máquinas e métodos organizados de modo a cumprir um certo número de funções específicas.

Concomitantemente à aplicação desse desenvolvimento surgiu a necessidade prática de administração nas grandes empresas ou nos complexos civis e militares, envolvendo o domínio de enormes quantidades de dados. Isso deu origem a diversos esforços em busca da racionalização e segurança desta tarefa.

A indústria eletrônica preparava o terreno para o que seria mais tarde chamada Segunda Revolução Industrial, isto é, a informatização da sociedade. Mas tudo começou com os computadores sendo tratados somente como equipamentos destinados à pesquisa científica ou, no máximo, à realização de cálculos estatísticos com fins militares.

Foram duas as principais transformações do mundo econômico e social, surgidas em decorrência da Revolução Industrial. A primeira com a concentração da produção e a multiplicação da capacidade produtiva do homem através do domínio da energia. E a segunda, o maior aproveitamento das máquinas e processos, cada vez mais concentrados e homogêneos.

Desta forma, a oficina artesanal desapareceu e deu lugar no início a uma pequena fábrica, que com o passar do tempo cresceu, assim como seu ambiente. Tornou-se impossível resolver todos os problemas no contexto da oficina artesanal e da pequena fábrica, que estavam ao alcance de uma ou duas pessoas. Daí a necessidade de alguém para cuidar das máquinas, alguém para a questão contábil e legal, alguém para as compras de matéria prima e alguém para tratar com os clientes.

Para que tudo funcionasse de maneira que agradasse a todos, todas estas pessoas participavam do processo produtivo, administrativo e comercial e deviam comunicar-se, uma vez que todo o trabalho dependia cada vez mais dos outros e dos fatores externos à empresa. As tarefas deviam ser harmonizadas, organizadas e divididas entre todos, com funções diferentes para cada um.

A comunicação era restrita, ou seja, cada indivíduo só podia trocar informações com poucos. Esta prática provocou a redução do tempo gasto para passar as informações, bem como para a assimilação das mesmas, muitas vezes excessivas e inúteis ou de pouca importância para o desempenho de uma tarefa.

Em função destes fatos surgiu o planejamento de organização definindo medidas complementares de comunicação (o que e a quem comunicar). Além disso, foram aprimoradas as regras sobre a comunicação escrita. Descobriu-se que a utilização de formulários impressos propiciava maior precisão de comunicação e economia de tempo (definido em administração como burocracia).

Apesar do controle mais rígido da comunicação de um indivíduo com outro, ocorria perda de informações importantes. Para economizar tempo era necessário comunicar e registrar apenas o essencial. Porém, como saber o que era importante? O que é essencial para uma função pode não ser para outra.

Outras formas de comunicação e controle foram estabelecidas. Novos formulários e regras se impuseram para se adaptar a nova realidade econômica.

A informação passou a ser administrada, independente de seu significado e uso. Tornou-se necessário planejar e definir procedimentos de tratamento, produção e controle dos fluxos de informações.

Nesse momento, começava a Segunda Revolução Industrial – eram usadas ideias de mecanização pelo uso da eletricidade para o processamento das informações, como calculadoras mecânicas, controle automático de equipamentos, e tabuladoras tipo "Hollerith" para armazenamentos de dados. Procuravam-se, ao mesmo tempo, métodos matemáticos e lógicos para avaliar situações complexas com que as empresas, mais evoluídas e de maior porte, defrontavam

Foi aí que surgiu a necessidade de ampliar a capacidade do homem em manusear e processar informações, fossem elas numéricas ou não, empresariais ou militares, detalhadas ou sintéticas.

Assim, foram estudadas as estruturas organizacionais, procurando visualizar e examinar os diversos mecanismos de interação entre as mesmas, enfatizando a dinâmica da informação.

Os estudos destas estruturas permitiram, através da análise de sistemas e sua consequente informatização, dominar a complexidade de empresas gigantescas que gradativamente iam surgindo a partir da década de 50.

Depois que passamos pela segunda Revolução Industrial entramos numa nova era. A partir da década de 90, com o advento das Novas Tecnologias de Informação e Comunicação (NTICs) propiciamos a formação da Sociedade da Informação ou do Conhecimento. A dita terceira Revolução Industrial ou a Era da Informação.



Estudo Complementar

Aproveite para conhecer o American National Standards Institute (ANSI) através do seu site: <http://www.ansi.org/>

e mais características deste importante órgão no Wikipédia (<http://pt.wikipedia.org/wiki/Ansi>).



Atividades

Descreva sucintamente como foi o início histórico da Análise de Sistemas e as três fases da Revolução Industrial



UNIDADE 2

Revisão Geral de Engenharia de Software

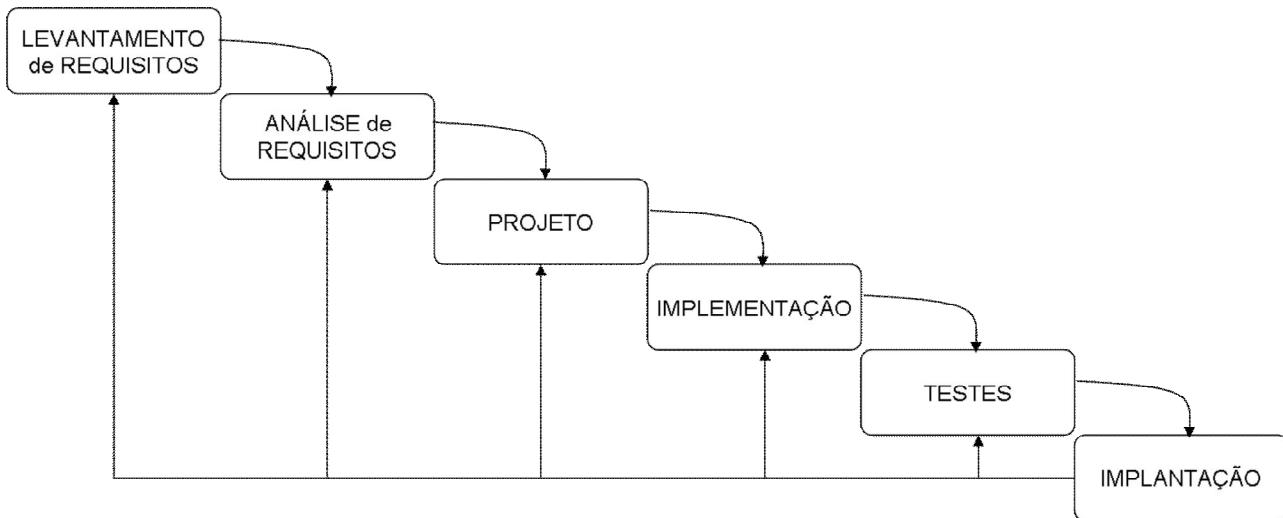
Objetivo: Revisar de forma sintética os principais conceitos de Engenharia de Software.

Vamos agora revisar a Engenharia de Software aos olhos da Metodologia de Análise de Sistemas. Um dos tópicos iniciais é o modelo de Ciclo de Vida. Podemos resumir os diversos modelos existentes na prática em dois: o “Modelo Cascata” e o “Modelo Iterativo e Incremental”.

Modelo Cascata

O Modelo Cascata também chamado de Clássico ou Linear se caracteriza por possuir uma tendência na progressão sequencial entre uma fase e a seguinte. Eventualmente, pode haver uma retroalimentação de uma fase para a fase anterior, mas de um ponto de vista macro, as fases seguem fundamentalmente de forma sequencial.

Os projetos de desenvolvimento reais raramente seguem o fluxo sequencial que esse modelo propõe. Tipicamente, algumas atividades de desenvolvimento podem ser realizadas em paralelo. E a entrega do sistema ao usuário, por esse modelo, somente ocorrerá no final do projeto. O que na maioria dos casos ocorre é que os requisitos iniciais foram alterados ou modificados e o sistema não vai mais corresponder à realidade, ou às necessidades do usuário.



Modelo Iterativo E Incremental

O Modelo de ciclo de vida Iterativo e Incremental foi proposto justamente para ser a resposta aos problemas encontrados no Modelo em Cascata. Um processo de desenvolvimento segundo essa abordagem divide o desenvolvimento de um produto de software em ciclos. Em cada ciclo de desenvolvimento, podem ser identificadas as fases de análise, projeto, implementação e testes. Essa característica contrasta com a abordagem clássica, na qual as fases de análise, projeto, implementação e testes são realizadas uma única vez.

No Modelo de ciclo de vida Iterativo e Incremental, um sistema de software é desenvolvido em vários passos similares (iterativo). Em cada passo, o sistema é estendido com mais funcionalidades (incremental).

A abordagem incremental incentiva a participação do usuário nas atividades de desenvolvimento do sistema, o que diminui em muito a probabilidade de interpretações erradas em relação aos requisitos levantados.

Outra vantagem dessa abordagem é que os riscos do projeto podem ser mais bem gerenciados. Um risco de desenvolvimento é a possibilidade de ocorrência de algum evento que cause prejuízo ao processo de desenvolvimento, juntamente com as consequências desse prejuízo. Os requisitos a serem considerados primeiramente devem ser selecionados com base nos riscos que eles fornecem. Os requisitos mais arriscados devem ser considerados, tão logo possível.

Para entender o motivo de que o conjunto de requisitos deve ser considerado o mais cedo possível, vamos nos lembrar de uma famosa frase do consultor Tom Gilb (1988): “Se você não atacar os riscos ativamente, então estes irão ativamente atacar você”. Ou seja, quanto mais cedo a equipe de desenvolvimento considerar os requisitos mais arriscados, menor é a probabilidade de ocorrerem prejuízos devido a esses requisitos.



Estudo Complementar

Conheça mais sobre o polêmico engenheiro de sistemas Tom Gilb no Wikipédia americano: http://en.wikipedia.org/wiki/Tom_Gilb



Atividades

Responda a questão: Quais são os diferenciais do Modelo de ciclo de vida Iterativo e Incremental quanto ao modelo clássico?

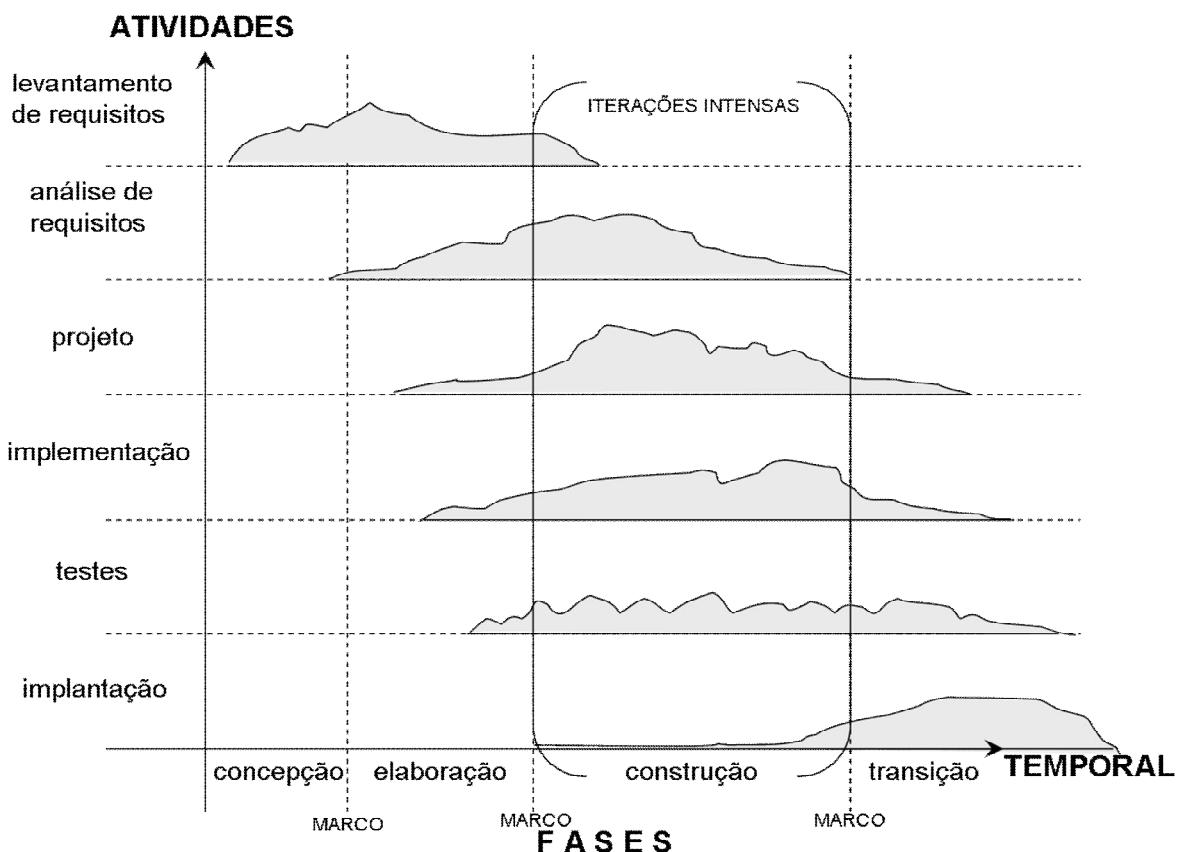


UNIDADE 3

Estrutura Geral de um Desenvolvimento Incremental e Iterativo

Objetivo: Detalhar e definir a estrutura geral de um Desenvolvimento Incremental e Iterativo.

Basicamente o ciclo de vida de processo Incremental e Iterativo pode ser estudado segundo duas dimensões: a temporal e a de atividades. Veja mais atentamente a figura abaixo:



A **dimensão temporal**, na horizontal (na parte mais inferior do gráfico), os processos estão estruturados em **FASES** (concepção, elaboração, construção e transição). Em cada uma dessas fases, há uma ou mais **iterações**. Cada iteração tem uma duração preestabelecida (de duas a seis semanas). Ao final de cada iteração, é produzido um incremento, ou seja,

uma parte do sistema final. Um incremento pode ser liberado para os usuários, ou pode ser somente um incremento interno.

A **dimensão de atividades** (ou de fluxos de trabalho) é apresentada verticalmente na figura anterior. Essa dimensão compreende as atividades realizadas durante a iteração de uma dessas fases: levantamento de requisitos, análise de requisitos, projeto, implementação, testes e implantação (as mesmas atividades que veremos mais detalhadamente adiante).

Em cada uma dessas fases, diferentes artefatos de software são produzidos, ou artefatos começados em uma fase anterior são estendidos com novos detalhes. Cada fase é concluída com um **MARCO**. Um marco é um ponto do desenvolvimento no qual decisões sobre o projeto são tomadas e importantes objetivos são alcançados. Os marcos são úteis para o gerente de projeto estimar os gastos e o andamento do cronograma de desenvolvimento.

Como vimos anteriormente as fases que são delimitadas pelos marcos são: **concepção, elaboração, construção e transição**. Vejamos agora com mais detalhes cada uma dessas fases:

CONCEPÇÃO:

Nessa fase a ideia geral, e o escopo do desenvolvimento, são desenvolvidos. Um planejamento de alto nível do desenvolvimento é realizado. São determinados os marcos que separam as fases.

ELABORAÇÃO

É alcançado um entendimento inicial sobre como o sistema será construído. O planejamento do projeto de desenvolvimento é completado. Nessa fase, o domínio do negócio é analisado. Os requisitos do sistema são ordenados considerando-se prioridade e risco. Também são planejadas as iterações da próxima fase, a de construção. Isso envolve definir a duração de cada iteração e o que será desenvolvido em cada iteração.

CONSTRUÇÃO

As atividades de análise e projeto aumentam em comparação às demais. Esta é a fase na qual ocorrem mais iterações incrementais. No final dessa fase, é decidido se o produto de software pode ser entregue aos usuários sem que o projeto seja exposto a altos riscos. Se este for o caso, tem início a construção do manual do usuário e da descrição dos incrementos realizados no sistema.

TRANSIÇÃO

Os usuários são treinados para utilizar o sistema. Questões de instalação e configuração do sistema também são tratadas. Ao final desta fase, a aceitação do usuário e os gastos são avaliados. Uma vez que o sistema é entregue aos usuários, provavelmente surgem novas questões que demandam a construção de novas versões do mesmo. Neste caso, um novo ciclo de desenvolvimento é iniciado.

Em cada iteração, uma proporção maior ou menor de cada dessas atividades é realizada, dependendo da fase em que se encontra o desenvolvimento. Na figura, com o intuito de mostrar um modelo amplo da estrutura geral de um Desenvolvimento Incremental e Iterativo, permite-se perceber, por exemplo, que na fase de transição, a atividade de implantação é a predominante. Por outro lado, na fase de construção, as atividades de análise, projeto e implementação são as de destaque. Normalmente, a fase de construção é a que possui mais interações. No entanto, as demais fases também podem conter iterações, dependendo da complexidade do sistema.

O principal representante da abordagem de desenvolvimento incremental e iterativo é o denominado **RUP - Rational Unified Process** (Processo Unificado Racional). Este processo de desenvolvimento é patenteado pela empresa Rational, onde trabalham os famosos três amigos (Jacobson, Booch e Rumbaugh). A descrição feita nesta seção é uma versão simplificada do Processo Unificado.

Veremos nas unidades a seguir um detalhamento de cada uma das fases apresentadas nesta aula.



Estudo Complementar

Aproveite para visitar o site da Rational

<http://www.rational.com/products/rup/index.jtmpl> e leia o artigo “Best Practices for Software Development Teams”.



Atividades

Tente reproduzir, somente de memória, o gráfico apresentado nesta unidade representando o modelo amplo da estrutura geral de um Desenvolvimento Incremental e Iterativo.



UNIDADE 4

Processo de Desenvolvimento de Software: Levantamento de Requisitos

Objetivo: Destacar as características e a importância, no desenvolvimento de software, da fase de levantamento de requisitos.

Para dar uma ideia da realidade atual no desenvolvimento de sistemas de software, são listados a seguir alguns dados levantados no Chaos Report (www.projectsmart.co.uk/docs/chaos-report.pdf), um estudo feito pelo Standish Group, sobre projetos de desenvolvimento:

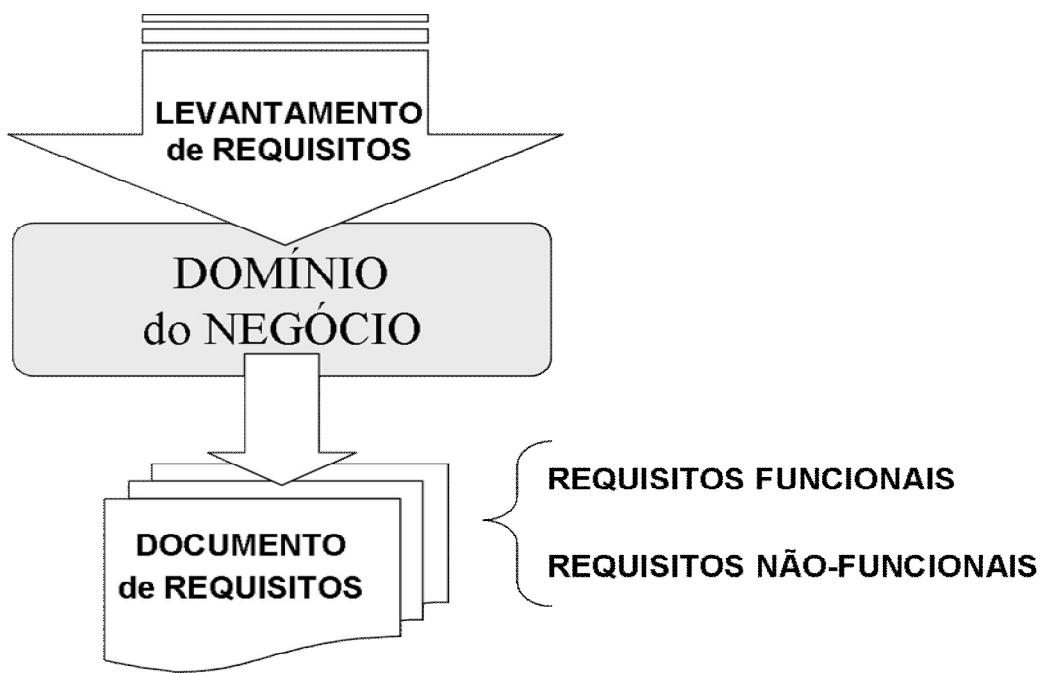
- Porcentagem de projetos que terminam dentro do prazo estimado: **10%**
- Porcentagem de projetos que são descontinuados antes de chegarem ao fim: **25%**
- Porcentagem de projetos acima do custo esperado: **60%**
- Atraso médio nos projetos: **1 ano**

Como já vimos em unidades anteriores um processo de desenvolvimento pode ser estruturado em atividades realizadas durante a construção de um sistema de software. E há vários processos de desenvolvimento propostos, no entanto não existe o melhor processo de desenvolvimento. Cada processo tem suas particularidades em relação ao modo de arranjar e encadear as atividades de desenvolvimento. Veremos a seguir as mais comuns e tradicionais atividades de um processo de desenvolvimento.

Levantamento De Requisitos

Pela definição de Maciaszek (2000), temos que **requisito** é uma condição ou capacidade que deve ser alcançada ou possuída por um sistema ou componente deste para satisfazer um contrato, padrão, especificação ou outros documentos formalmente impostos.

Normalmente os requisitos de um sistema são identificados a partir do **domínio do negócio**. Denomina-se domínio de negócio a área de conhecimento específica na qual um determinado sistema de software será desenvolvido. Ou seja, domínio de negócio corresponde à parte do mundo real que é relevante ao desenvolvimento de um sistema. O domínio de negócio também é chamado de domínio do problema ou domínio da aplicação. Veja a figura a seguir:



Durante o levantamento de requisitos, a equipe de desenvolvimento tenta entender o domínio do negócio que deve ser automatizado pelo sistema de software. O levantamento de requisitos compreende um estudo exploratório das necessidades dos usuários e da situação do sistema atual (se esse existir). Alguns autores aconselham ao analista a não perder tempo com o sistema atual, e partir diretamente para a concepção do novo. Este conselho se deve ao fato de o analista não ficar “amarrado” aos conceitos da estrutura antiga, e poder ser mais inovador possível em sua nova proposta.

O produto final do levantamento de requisitos é o Documento de Requisitos, que declara os diversos tipos de requisitos do sistema. Normalmente esse documento é escrito em linguagem natural (notação informal). As principais seções de um documento de requisitos são:

REQUISITOS FUNCIONAIS

Definem as funcionalidades do sistema. Veremos mais adiante nesta apostila que tipicamente os Requisitos Funcionais serão alvo na UML, dos modelos de Caso de Uso. Alguns exemplos práticos de requisitos funcionais são:

“o sistema deve permitir que cada professor realize o lançamento de notas das turmas nas quais lecionou”

“o sistema deve permitir que o aluno realize a sua matrícula nas disciplinas oferecidas em um semestre”

“os coordenadores de escola devem poder obter o número de aprovações, reprovações e trancamentos em todas as turmas em um determinado período”

REQUISITOS NÃO-FUNCIONAIS

Declaram as características de qualidade que o sistema deve possuir e que estão relacionadas às suas funcionalidades. Para você visualizar melhor esse conceito, alguns tipos de requisitos não funcionais são relacionados a seguir:

CONFIABILIDADE

Corresponde a medidas quantitativas da confiabilidade do sistema, tais como tempo médio entre falhas, recuperação de falhas ou quantidade de erros por milhares de linhas de código-fonte.

DESEMPENHO

Requisitos que definem tempos de respostas esperados para as funcionalidades do sistema.

PORATIBILIDADE

Facilidade para transportar o sistema para outras plataformas.

SEGURANÇA

Limitações sobre a segurança do sistema em relação a acessos não autorizados.

USABILIDADE

Requisitos que se relacionam ou afetam a usabilidade do sistema. Exemplos incluem requisitos sobre a facilidade de uso e a necessidade ou não de treinamento dos usuários.

Uma das formas de se medir a qualidade de um sistema de software é através de sua utilidade. E um sistema será útil para seus usuários se atender aos requisitos definidos.

O enfoque prioritário do levantamento de requisitos é responder claramente a questão: “O que o usuário necessita do novo sistema?”. Requisitos definem o problema a ser resolvido pelo sistema de software; eles não descrevem o software que resolve o problema.

Lembre-se sempre: novos sistemas serão avaliados pelo seu grau de conformidade aos requisitos, não quanto complexa a solução tecnológica tenha sido aplicada.

O levantamento de requisitos é a etapa mais importante em termo de retorno de investimento feito para o projeto de desenvolvimento. Muitos sistemas foram abandonados ou nem chegaram a serem usados porque os membros da equipe não dispensaram tempo suficiente para compreender as necessidades do cliente. Em um estudo baseado em 6.700 sistemas feitos em 1997, Carper Jones mostrou que os custos resultantes da má realização desta etapa de entendimento podem ser 200 vezes maiores que o realmente necessário.

O Documento de Requisitos serve como um termo de consenso entre a equipe técnica de desenvolvedores e o cliente. Esse documento constitui a base para as atividades subsequentes do desenvolvimento do sistema e fornece um ponto de referência para qualquer validação futura do software construído. O envolvimento do cliente desde o início do processo de desenvolvimento ajuda a assegurar que o produto desenvolvido realmente atenda às necessidades identificadas.

Além disso, o Documento de Requisitos estabelece o **escopo do sistema**, isto é, o que faz parte e o que não faz parte do sistema. O escopo de um sistema muitas vezes muda durante o seu desenvolvimento. Desta forma, se o escopo muda, tanto clientes quanto

desenvolvedores têm um parâmetro para decidirem o quanto de recursos de tempo e financeiros devem mudar.

Outro ponto importante sobre requisitos é a sua característica de volatilidade. Um requisito volátil é aquele que pode sofrer modificações durante o desenvolvimento do sistema.

A menos que o sistema a ser desenvolvido seja bastante simples e estático, características cada vez mais raras nos sistemas atuais, é humanamente impossível pensar em todos os detalhes em princípio. Além disso, quando o sistema entrar em produção e os usuários começarem a utilizá-lo, eles próprios descobrirão requisitos nos quais nem sequer tinham pensado anteriormente.

Em resumo, os requisitos de um sistema complexo inevitavelmente mudarão durante todo o seu desenvolvimento. No desenvolvimento de sistemas de software, a existência de requisitos voláteis corresponde mais a regra do que à exceção.



Estudo Complementar

Visite o site do Prof. Maciaszek: <http://www.comp.mq.edu.au/~leszek/>

http://www.branqs.com.br/universidade/aulas_EngSoft_Ciencias/0002_LevantamentoDeRequisitos/LevantamentoDosRequisitosDoSistema.pdf

<http://www2.iesam-pa.edu.br/pids/descrs/DescLevantamentoRSw.html>



Atividades

Baseado em sua experiência acadêmica, tente desenvolver um Documento de Requisitos para um imaginável Sistema de Controle Universitário. Esse sistema deve controlar as funções básicas de uma faculdade tais como: controle das inscrições de alunos em disciplinas, alocação de turmas, salas e professores e assim por diante. Deve permitir também o controle de notas atribuídas aos alunos nas diversas disciplinas.

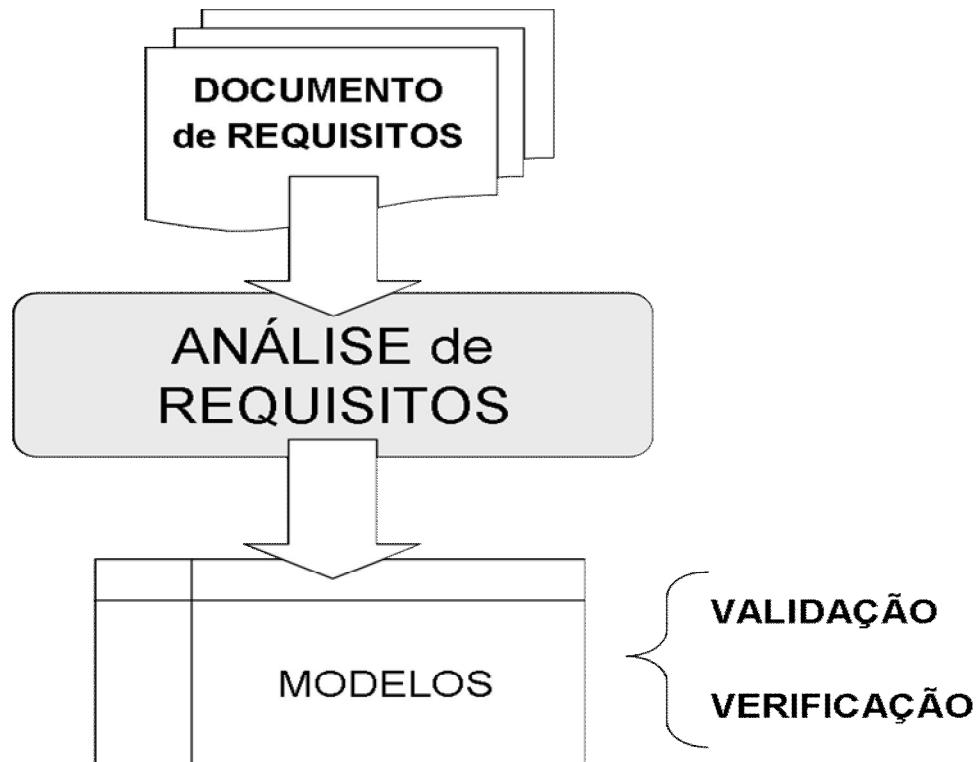


UNIDADE 5

Processo de Desenvolvimento de Software: Análise de Requisitos

Objetivo: Descrever os principais pontos da fase de Análise de Requisitos no processo de desenvolvimento de software.

Formalmente, o termo **análise** corresponde a quebrar um sistema em seus componentes, e estudar como tais componentes interagem com o objetivo de entender como esse mesmo sistema funciona. No contexto dos sistemas de software, esta é a etapa na qual os analistas realizam um estudo detalhado no Documento de Requisitos levantado na atividade anterior. A partir desse estudo, são construídos modelos para representar o sistema a ser construído (veja mais detalhes na figura a seguir). A Análise de Requisitos é também chamada por alguns autores como Especificação de Requisitos.



Assim como no levantamento de requisitos, a Análise de Requisitos não leva em conta o ambiente tecnológico a ser utilizado. Nessa atividade, o foco de interesse é tentar construir uma estratégia de solução, sem se preocupar com a maneira como essa estratégia será realizada.

A razão dessa prática é tentar obter a melhor solução para o problema sem se preocupar com os detalhes da tecnologia a ser utilizada. É necessário saber o que o sistema proposto precisa fazer para então, definir como esse sistema irá fazê-lo.

Ocorre que, frequentemente na prática, as equipes de desenvolvimento passam para a construção da solução sem antes terem definido completamente o problema. Portanto, os modelos construídos nesta fase devem ser cuidadosamente validados e verificados, através da validação e verificação dos modelos respectivamente.

O objetivo da validação é assegurar que as necessidades do cliente estão sendo atendidas pelo sistema: “será que o software correto está sendo construído?”.

Já a verificação tem o objetivo de verificar se os modelos construídos estão em conformidade com os requisitos definidos: “será que o software está sendo construído corretamente?”.

Na verificação dos modelos, são analisadas a exatidão de cada modelo em separado e a consistência entre os modelos.

Em um processo de desenvolvimento orientado a objetos, o resultado da análise são modelos que representam as estruturas das classes de objetos que serão componentes do sistema. Além disso, a análise também resulta em modelos que especificam as funcionalidades do sistema de software.

Prototipagem

A construção de protótipos é uma técnica que serve de complemento à análise de requisitos. No contexto do desenvolvimento de software, um protótipo é um esboço de alguma parte do sistema.

Protótipos são construídos para telas de entrada, telas de saída, subsistemas, ou mesmo para os sistemas como um todo. A construção de protótipos utiliza as denominadas

linguagens de programação visual. Exemplos são o Delphi, o PowerBuilder e o Visual Basic que, na verdade, são ambientes com facilidades para a construção da interface gráfica (telas, formulários, etc.). Além disso, muitos Sistemas de Gerência de Bancos de Dados também fornecem ferramentas para a construção de telas de entrada e saída de dados.

Na prototipagem, após o levantamento de requisitos, um protótipo do sistema é construído para ser usado na validação. O protótipo é revisto por um ou mais usuários, que fazem suas avaliações e críticas acerca das características apresentadas. O protótipo é então corrigido ou refinado de acordo com as intervenções dos usuários.

Esse processo de revisão e refinamento continua até que o protótipo seja aceito pelos usuários. Portanto, a técnica de prototipagem é muito útil e tem o objetivo de assegurar que os requisitos do sistema foram realmente bem entendidos.

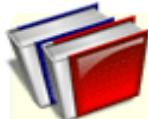
O resultado da validação através do protótipo pode ser usado para refinar os modelos do sistema. Após a aceitação, o protótipo (ou parte dele) pode ser descartado ou utilizado como uma versão inicial do sistema.

Embora a técnica de prototipagem seja opcional, ela é frequentemente aplicada em projetos de desenvolvimento de software, especialmente quando há dificuldades no entendimento dos requisitos do sistema, ou há requisitos arriscados que precisam ser mais bem entendidos.

A ideia é que um protótipo é mais concreto para fins de validação do que modelos representados por diagramas bidimensionais (tipo DFD, ou mesmo o UML). Isso incentiva a participação ativa do usuário na validação. Consequentemente, a tarefa de validação se torna menos suscetível a erros. No entanto, destacamos que alguns desenvolvedores usam essa técnica como um substituto à construção de modelos de sistema.

Tenha em mente que a prototipagem é uma técnica complementar à construção dos modelos do sistema. Os modelos do sistema devem ser construídos, pois são eles que guiam as demais fases do projeto de desenvolvimento de software.

Idealmente, os erros detectados na validação do protótipo devem ser utilizados para modificar e refinar os modelos do sistema. Portanto, devemos utilizar a prototipagem como complemento ao Modelo de Ciclo de Vida Iterativo e Incremental, e não para substituí-la.



Estudo Complementar

Veja o interessante texto, no Wikipédia, sobre o uso da prototipagem:
http://pt.wikipedia.org/wiki/Uso_da_Prototipagem_na_Eng._de_Requisitos



Atividades

Veja o blog <http://maozinhadaweb.blogspot.com/2007/05/anlise-de-requisitos-funcionais-x-no.html> e realize uma análise crítica do aluno da UFB.



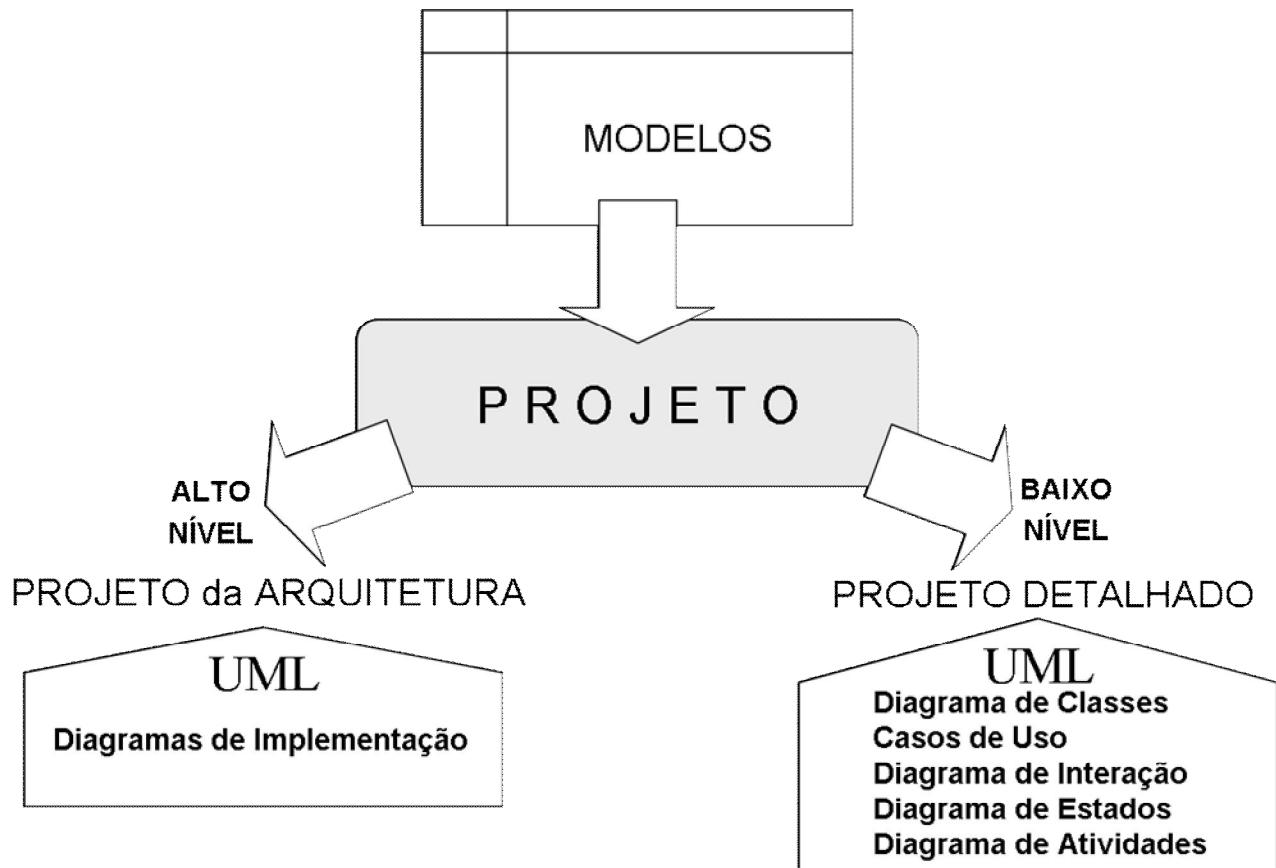
UNIDADE 6

Processo de Desenvolvimento de Software: Projeto, Implementação, Testes e Implantação

Objetivo: Apresentar as últimas etapas e a relação das mesmas no Processo de Desenvolvimento de Software.

O foco principal da análise são os aspectos lógicos e independentes de implementação de um sistema, os requisitos. Na fase de Projeto, determina-se “como” o sistema funcionará para atender aos requisitos, de acordo com os recursos tecnológicos existentes – a fase de projeto considera os aspectos físicos e dependentes de implementação.

Aos modelos construídos na fase de análise são adicionadas as determinadas “restrições de tecnologia”. Exemplos de aspectos a serem considerados na fase de projeto: arquitetura do sistema, padrão de interface gráfica, a linguagem de programação, o Gerenciador de Banco de Dados, etc.



Esta fase produz uma descrição computacional do que o software deve fazer, e deve ser coerente com a descrição feita na análise. Em alguns casos, algumas restrições da tecnologia a ser utilizada já foram amarradas no Levantamento de Requisitos. Em outros casos, essas restrições devem ser especificadas. Mas, em todos os casos, a fase de projeto do sistema é direcionada pelos modelos construídos na fase de análise e pelo planejamento do sistema.

O projeto consiste de duas atividades principais: **Projeto da Arquitetura**, também conhecido como projeto de alto nível, e **Projeto Detalhado** denominado também como projeto de baixo nível.

Em um processo de desenvolvimento orientado a objetos, o Projeto da Arquitetura consiste em distribuir as classes de objetos relacionados do sistema em subsistemas e seus componentes. Consiste também em distribuir esses componentes fisicamente pelos recursos

de hardware disponíveis. Os diagramas da UML normalmente utilizados nesta fase do projeto são os **Diagramas de Implementação**.

No Projeto Detalhado, são modeladas as colaborações entre os objetos de cada módulo com o objetivo de realizar as funcionalidades do módulo. Também são realizados o projeto da interface com o usuário e o projeto de Banco de Dados. Os principais diagramas da UML utilizados nesta fase do projeto são: **Diagrama de Classe**, **Diagrama de Casos de Uso**, **Diagrama de Interação**, **Diagrama de Estados** e **Diagrama de Atividades**.

Embora a Análise e o Projeto sejam descritos didaticamente em seções separadas, é importante notar que na prática não há uma distinção tão clara entre essas duas fases. Principalmente no desenvolvimento de sistemas orientados a objetos, as atividades dessas duas fases frequentemente se misturam.

IMPLEMENTAÇÃO

Na Implementação, o sistema é codificado, ou seja, ocorre a tradução da descrição computacional da fase de projeto em código executável através do uso de uma ou mais linguagens de programação.

Em um processo de desenvolvimento orientado a objetos, a implementação envolve a definição das classes de objetos do sistema utilizando linguagens de programação como C++, Java, etc. Além da codificação desde o início, a Implementação pode e deve também utilizar componentes de software e bibliotecas de classes preexistentes para agilizar a atividade.

TESTES

Diversas atividades de teste são realizadas para verificação do sistema construído, levando-se em conta a especificação feita na fase de Projeto. O principal produto dessa fase é o Relatório de Testes, contendo informações sobre erros detectados no software. Após a atividade de testes, os diversos módulos do sistema são integrados, resultando finalmente no produto de software.

IMPLEMENTAÇÃO

O sistema é empacotado, distribuído e instalado no ambiente do usuário. Os manuais do sistema são escritos, os arquivos são carregados, os dados são importados para o sistema e os usuários são treinados para utilizar o sistema corretamente. Em alguns casos, principalmente em sistemas legados, aqui também ocorre a migração de sistemas de software e de dados preexistentes.



Atividades

Reflita e responda as seguintes perguntas:

1. Como estas últimas quatro fases do desenvolvimento de software se relacionam com as primeiras etapas?
2. Como você classificaria todas essas fases em ordem de importância?



UNIDADE 7

Objetivo: Conceituar metodologia e relacionar com a Análise de Sistemas dentro do contexto da Engenharia de Software.

O que é Metodologia? O que é Análise de Sistemas? O contexto dentro de Engenharia de Software

Iremos começar esta unidade pela frase do Prof. Jayr Figueiredo (<http://donjf.sites.uol.com.br/>) que retrata bem a realidade da implantação de metodologias na área de Sistemas:

“Quando você tentar implantar uma nova metodologia para desenvolvimento de sistemas, certamente irá cometer erros. Porém, se não tentar implantá-la, já estará errando”.

Para nos aprofundarmos nesse conceito precisamos distinguir as palavras “Método” e “Metodologia”. Embora utilizada indiscriminadamente tanto de uma forma com de outra, existe diferença significativa entre essas palavras.

Método

É o caminho pelo qual se atinge um objetivo. Pode-se definir também como um programa que regula previamente uma série de operações que se devem realizar, apontando erros evitáveis, em vista de um resultado determinado. Com isso, temos que o método é o caminho ordenado e sistemático para chegar a um fim.

Metodologia

É a arte de dirigir o espírito na investigação da verdade, ou simplesmente como o “estudo dos métodos” e, especialmente, dos métodos da ciência. Consiste em avaliar, analisar e estudar os vários métodos disponíveis pela emissão e aprovação das técnicas, as quais serão aplicadas futuramente, oferecendo algumas formas de divulgação que orientem outras aplicabilidades.

Portanto, a Metodologia é mais ampla que Método, pois a primeira estuda a segunda. Dentro desse contexto a metodologia pode ser considerada como um sistema para desenvolver sistemas. Devemos lembrar que nem sempre a metodologia de trabalho adotada pode trazer benefícios e os resultados esperados pelas empresas. Principalmente se a metodologia apresentar uma filosofia, uma política e uma estrutura eficazes, mas pouco eficiente ou muito burocrática.

Por outro lado, uma metodologia bem implantada pode propiciar a qualquer empresa vários benefícios. Vejamos a seguir essas principais vantagens.

Benefícios De Uma Metodologia Bem Implantada

Aumento Da Qualidade Dos Sistemas

Os desenvolvedores têm à sua disposição métodos que permitem levantar com precisão as necessidades dos usuários e construir sistemas melhores estruturados. O uso de uma notação padronizada melhora a comunicação com os usuários e entre os próprios profissionais de sistemas. Portanto, é fundamental documentar.

Independência Dos Analistas

Como os sistemas são bem documentados e estruturados, um analista consegue dar manutenção a um sistema que não conheça previamente. Evita-se o erro da criação do “dono do sistema”, situação indesejável tanto para a empresa, para os usuários e para o próprio analista. Esse analista proprietário do sistema, além de não poder evoluir para outros desafios na empresa, fica tão amarrado ao seu próprio sistema que, muitas vezes, nem férias consegue tirar.

Facilidade De Manutenção

Complemento ao item anterior, destacando-se que com manutenções mais fáceis e rápidas, sobra mais tempo para desenvolver sistemas novos, ou a reengenharia dos antigos.

Aumento Da Produtividade

Sistemas bem construídos têm mais partes reutilizáveis. E, como o sistema é bem especificado e projetado, gastam-se menos tempo em testes e “gambiarras” (emendas) para atender ao usuário.

Um dos pontos que uma boa Metodologia aborda são as Métricas de Software. Podem-se definir Métricas de Software a uma ampla variedade de medidas de software, bem como, orientadas ao tamanho de medidas diretas do software e do processo por meio do qual ele é desenvolvido. Citando Pressman quando ao seu conceito de métricas temos:

“Quando se pode medir aquilo sobre o qual se está falando e expressá-lo em números, sabe-se alguma coisa sobre o mesmo; mas quando não se pode medi-lo, quando não se pode expressá-lo em números, o conhecimento que se tem é de um tipo inadequado e insatisfatório; este pode ser o começo do conhecimento, mas dificilmente alguém terá avançado em suas idéias para o estágio da ciência”.

Principais Objetivos Da Metodologia

Criar uma ferramenta que possibilite a desenvolvimento de projetos na empresa, em harmonia com os princípios elementares da administração, tais como: planejamento, previsão, organização, decisão, comando, coordenação e controle;

Promover o cumprimento de prazos, eficiência e qualidade do serviço, visando uma maior produtividade através da padronização das atividades de desenvolvimento e da racionalização dos controles e dos itens de documentação;

Servir de apoio ao desenvolvimento de projetos em suas etapas, orientando a execução das atividades requeridas em todos os níveis e setores envolvidos, de uma forma padronizada e integrada;

Estabelecer uma estrutura de documentação padronizada e compatível com a organização das fases e necessidades operacionais.



Estudo Complementar

Visite o site do Prof. Jayr Figueiredo: <http://donjf.sites.uol.com.br/>

http://searchsoftwarequality.techtarget.com/generic/0,295582,sid92_gci1249454,00.html?track=NL-776&ad=587562&Offer=SWQmod425lg&asrc=EM_USC_1352054



Atividades

Pesquise em sua empresa, ou na de colegas, o uso de metodologias. Que metodologias são determinadas pela empresa e quais são as efetivamente utilizadas por todos? Existe alguma punição no caso do não cumprimento dessas metodologias? Quais são os principais motivos do não cumprimento das metodologias?



UNIDADE 8

Objetivo: Especificar as principais funções de um Analista de Sistema e as principais Metodologias aplicadas

O que faz um Analista de Sistemas? A evolução das Metodologias e ferramentas CASE

Analista de Sistemas é o profissional que deve ter conhecimento do domínio do negócio. Esse profissional deve entender os problemas do domínio do negócio para que possa definir os requisitos do sistema a ser desenvolvido. Especialistas do Domínio são as pessoas que tem familiaridade com o domínio do negócio, mas não necessariamente com o desenvolvimento de sistemas de Software. Frequentemente, estes especialistas são os futuros usuários do Sistema em desenvolvimento.

Analistas devem estar aptos a se comunicar com especialistas do domínio para obter conhecimento acerca dos problemas e das necessidades envolvidas da organização empresarial. O Analista não precisa ser um especialista do domínio. Contudo, ele deve ter suficiente domínio do vocabulário da área de conhecimento na qual o sistema será implantado. Isso evita que ao se comunicar com o especialista de domínio, este não precise ser interrompido a todo o momento para explicar conhecimentos básicos da área.

Tipicamente, o Analista de Sistemas é o profissional responsável por entender as necessidades dos clientes em relação ao sistema a ser desenvolvido e repassar esse entendimento aos demais desenvolvedores de sistemas. Nesse sentido, o Analista de Sistemas representa a ponte de comunicação entre duas facções: a dos profissionais de computação e a dos profissionais do negócio.

Para realizar suas funções, o Analista de Sistemas deve entender não só do domínio do negócio da organização, mas também ter conhecimento dos aspectos mais complexos da computação. Nesse sentido, o Analista de Sistemas funciona como um tradutor, que mapeia informações entre duas “línguagens” diferentes: a dos especialistas de domínio e a dos profissionais técnicos da equipe de desenvolvimento.

Com a experiência adquirida através da participação no desenvolvimento de diversos projetos, alguns analistas se tornam gerentes de projetos. Na verdade, as possibilidades de evolução da carreira de um analista são bastante grandes. Isso se deve ao fato de que, durante a fase de levantamento de requisitos de um sistema, o analista se torna quase um especialista no domínio do negócio da organização. Para a organização, é bastante interessante ter em seu quadro um profissional que entenda ao mesmo tempo de técnicas de desenvolvimento de sistemas e do processo de negócio da empresa. Por essa razão, não é rara a situação em que uma organização oferece um contrato de trabalho ao analista de sistemas terceirizado ao final do desenvolvimento.

Uma característica importante que um analista de Sistemas deve ter é a capacidade de comunicação, tanto escrita como falada. Ele é um agente facilitador da comunicação entre os clientes e a equipe técnica. Muitas vezes, as capacidades de comunicar agilmente e de ter um bom relacionamento interpessoal são mais importantes para o analista do que o conhecimento tecnológico.

Outra característica necessária a uma analista é a ética profissional. Muitas vezes, o analista de sistemas está em contato com informações sigilosas e estratégicas dentro da organização na qual está trabalhando. Os analistas de Sistema têm acesso a informações como preços de custo de produtos, margens de lucro, algoritmos proprietários, etc.

Certamente, pode ser desastroso para a organização se informações de caráter confidencial como estas caírem em mãos erradas. Portanto, a ética profissional do analista de sistemas na manipulação de informações como essas é fundamental.

Os mandamentos do Analista de Sistemas e/ou Responsável pelo Projeto

- Estabeleça claramente quem é o líder do projeto, e defina cuidadosamente a organização do projeto.
- Enfatize sempre a importância do projeto como um todo, e não apenas de algumas etapas, fases ou aspectos.
- Dissipe os mitos contraproducentes sobre desenvolvimento de software.

- Forneça sempre feedback para a sua equipe, e principalmente aos seus usuários. Todos gostam de uma satisfação de como está a situação.
- Transpire entusiasmo, sempre. Pode parecer um pouco piegas, mas todos estarão vendo o seu desempenho no projeto e é importante mostrar o seu envolvimento.
- Crie visibilidade no projeto. Planeje reuniões de follow-up para que todos os envolvidos, os stakeholders, estejam a par do que está acontecendo.
- Envolva sempre a equipe nas atividades de planejamento, mesmo que tudo já esteja planejado.
- Seja sensível às diferenças pessoais de estilo, de habilidades, de motivações e de problemas da sua equipe, e de seus usuários.
- Equilibre liderança orientada para produção pela liderança orientada para pessoas.
- Aprenda a priorizar e re-priorizar rapidamente.

Evolução Das Metodologias De Desenvolvimento De Sistemas

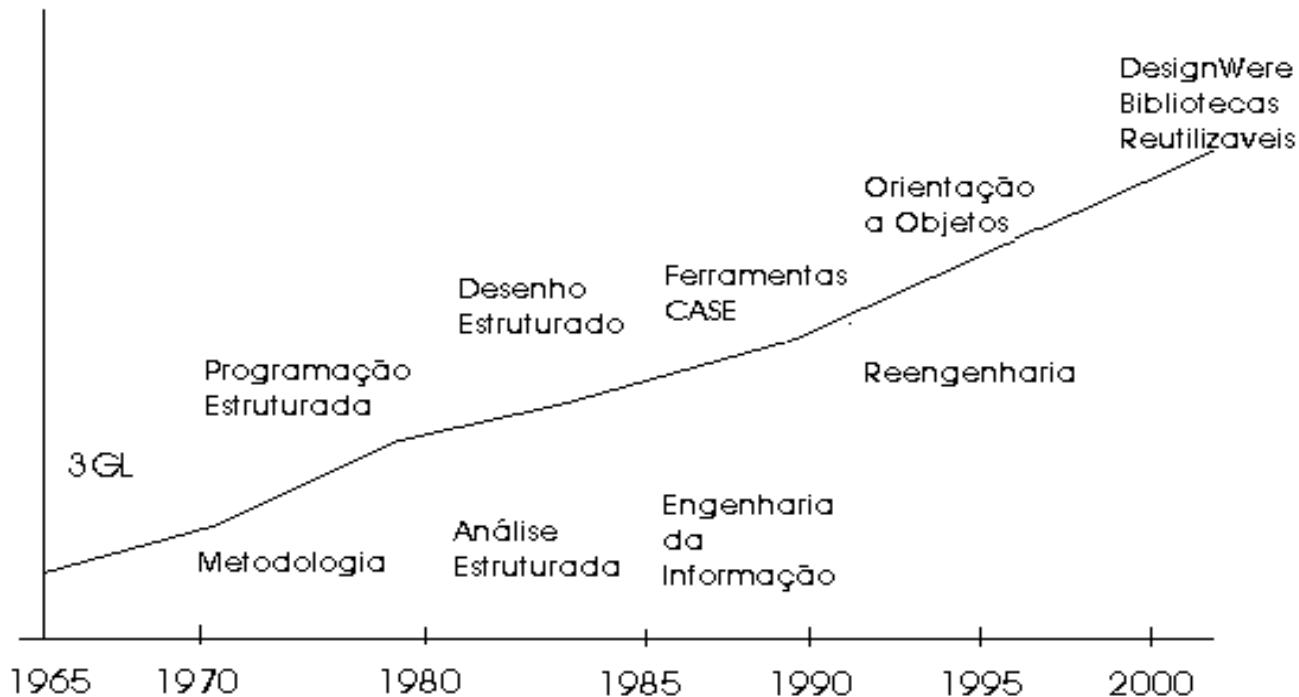
Década de 50	<ul style="list-style-type: none"> -Sistemas de baixa complexidade -Sistemas desenvolvidos sem planejamento -Fluxogramas -Diagramas de Módulos
Década de 60	<ul style="list-style-type: none"> -Início da Metodologia Estruturada
Década de 70	<ul style="list-style-type: none"> -Técnicas de Modelagem de Dados -Projeto de Banco de Dados -Banco de Dados

Década de 80	<ul style="list-style-type: none"> -Especificação do Projeto -Ferramentas de Software -Modelagem de Dados -Linguagens de quarta geração -Prototipação -Interface com o usuário -Automação das Metodologias
Década de 90	<ul style="list-style-type: none"> -Ferramentas de Geração de Código -Metodologias Orientadas a Objetos (MOO)
Final da Década de 90	<ul style="list-style-type: none"> -Maturidade da MOO -Início da UML

Como vimos, a Metodologia é a reunião de normas, métodos, controles, ferramentas, técnicas e políticas em uso numa empresa. Portanto, um conjunto muito grande de materiais e informações, visando a sua perfeita compreensão e utilização.

A metodologia definida pelas organizações não deve obrigar o uso de uma ou de outra técnica em especial. Deve estimular os profissionais a procurarem a técnica mais adequada ao problema a ser resolvido.

Cabe ao analista a escolha do ferramental específico dentro da orientação que é dada pela Metodologia, baseada em normas internacionais e reconhecida pela Engenharia de Software.



Ferramentas CASE

Um processo de desenvolvimento de software é altamente complexo. Várias pessoas, com diferentes especialidades, estão envolvidas neste processo altamente cooperativo. Tal atividade cooperativa pode ser facilitada através de uso de ferramentas que auxiliam na construção de modelos de sistema, na integração do trabalho de cada membro da equipe, no gerenciamento do andamento do desenvolvimento, etc.

Sistemas de software que são utilizados para dar suporte ao ciclo de vida de desenvolvimento são normalmente chamados de ferramentas CASE. O termo CASE é uma sigla em inglês para Engenharia de Software auxiliada por Computador (Computer Aided Software Engineering). A utilização dessa sigla já se consolidou no Brasil.

A Metodologia é a base e CASE é a automação da metodologia. As ferramentas CASE são uma combinação de utilitários de software com a Metodologia. Enfoca o problema da produtividade e não somente os problemas de implementação. Para a sua implementação

são necessárias a inclusão de treinamento intensivo e a seleção de grupo experiente para esse treinamento.

Como principais características dessas ferramentas podem-se destacar a Engenharia Reversa, repositórios (mecanismo para armazenamento e organização de todas as informações concernentes ao sistema) e software reusável.

Enfim, a automação das metodologias tornou-se uma constante nos sistemas. Otimiza e evita o uso da programação manual. Com a tecnologia CASE, incrementa-se o processo sistêmico computacional em poder e capacidade.

Existem diversas ferramentas CASE disponíveis no mercado. Algumas das características que podem ser encontradas em ferramentas CASE são sucintamente descritas a seguir:

Diagramas

Criação de diagramas e manutenção da consistência entre esses diagramas;

Round-trip engineering

Geração de código-fonte a partir de diagramas e geração de diagramas a partir do código-fonte;

Depuração do código-fonte

Ferramentas que permitem encontrar erros de lógica em partes de um programa;

Relatórios de testes

Ferramentas que geram relatório informando sobre partes de um programa que não foram testadas.

Testes automáticos

Ferramentas que realizam testes automaticamente no sistema;

Gerenciamento de versões

Ferramentas que permitem gerenciar as diversas versões dos artefatos de software gerados durante o ciclo de vida de um sistema;

Verificação de desempenho

Averiguar o tempo de execução de módulos de um sistema, assim como o tráfego de dados em sistemas de rede;

Erros

Verificação de erros em tempo de execução;

Mudanças

Gerenciamento de mudanças nos requisitos.

Alem das ferramentas CASE, outras ferramentas importantes em um processo de desenvolvimento são as que fornecem suporte ao gerenciamento. Essas ferramentas são utilizadas pelo Gerente de Projeto para desenvolver atividades tais como: cronogramas de tarefas, definição de alocações de verbas, monitoramento do progresso e os gastos, geração de relatórios de gerenciamento.



Estudo Complementar

http://pt.wikipedia.org/wiki/Ferramenta_CASE

http://pt.wikipedia.org/wiki/An%C3%A1lise_de_sistemas#

<http://br.answers.yahoo.com/question/index?qid=20061208183234AAXCP6A>

http://www.timaster.com.br/revista/artigos/main_artigo.asp?codigo=1308

<http://www.timaster.com.br/revista/raiox/raiox.asp>

<http://www.brasilprofissoes.com.br/verprof.php?codigo=512>



Atividades

Leia o interessante artigo da Palloma P. Souza, e também o fórum sobre a discussão entre as diferenças entre o Analista de Sistema e o de Negócios:

<http://osdir.com/ml/education.brazil.infoestacio/2006-10/msg00234.html>

<http://www.guj.com.br/posts/list/71342.java>



UNIDADE 9

Objetivo: Apresentar historicamente as principais metodologias aplicadas a desenvolvimento de sistemas.

Metodologias de Análise de Sistemas: Visão Geral

As metodologias de sistemas como vimos são utilizadas para estabelecer ordem, definir padrões e usar técnicas já provadas no desenvolvimento de sistemas, agilizando o processo e garantindo maior qualidade no desenvolvimento.

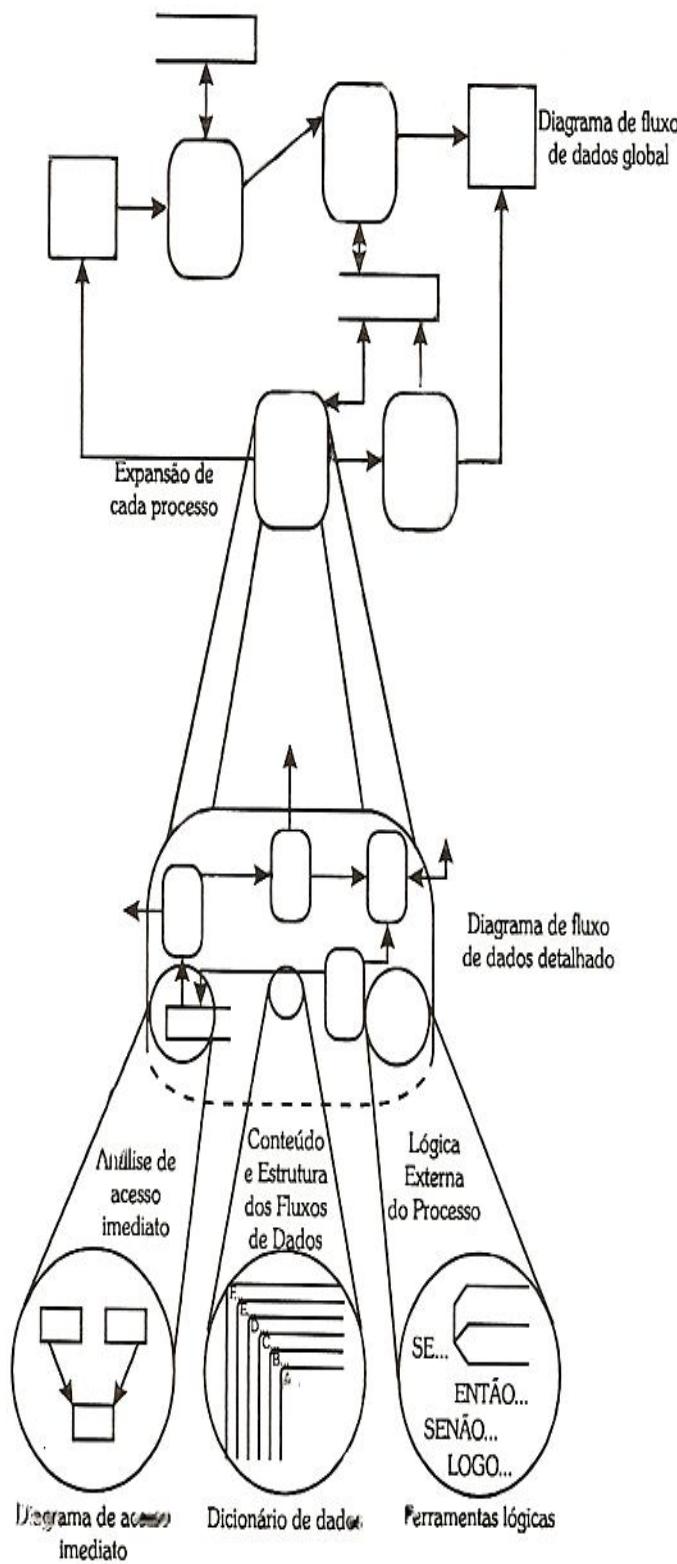
Atualmente existem três tipos de metodologias que se destacam: a **estruturada**, a de **desenvolvimento ágil** e a **orientada por objetos**. As diferenças nas metodologias estão nas técnicas de construção do processo de negócio, as definições dos dados e os modelos de eventos.

Para apoiar as metodologias foram criadas ferramentas para acompanhar o ciclo de vida do sistema, auxiliando no desenvolvimento de aplicativos (como as ferramentas CASE, visto na unidade anterior) e no gerenciamento do projeto. Dentre as ferramentas mais utilizadas está o RAD (Rapid Application Development), onde são utilizadas sessões de planejamento com os usuários para definir o sistema de aplicação.

As metodologias para desenvolvimento de sistemas devem acompanhar todo o ciclo de vida dos sistemas. Todas as metodologias usam gráficos para representar os elementos de sistemas. E as descrições e definições de cada elemento são relacionadas no diagrama.

Metodologia Estruturada

Na metodologia estruturada de análise e desenvolvimento de sistemas (SSAD – Structured System Analysis and Design) estabelecem-se sucessivos detalhamentos dos processos desde o nível macro, até o detalhe de mais baixo nível. É uma visão top-down de sistemas. Essa é a metodologia mais antiga, e ainda em uso por várias instituições.



Para representar os elementos de sistemas na Metodologia Estruturada se usa o DFD (Data Flow Diagram). Os DFDs descrevem o fluxo de dados no sistema. Cada Diagrama de Fluxo de Dados - DFD incorpora mais detalhes do processo fazendo uma explosão de cada componente do diagrama.

O diagrama DER (Entity Relation Diagram - Diagrama Entidade Relacionamento) de relacionamento de entidades (por exemplo: um objeto, uma pessoa, etc.) normaliza os dados do ambiente e define os dados para a aplicação. Em um diagrama de modelo de dados com base no relacionamento entre as entidades definem-se os registros estruturados para serem definidos nas bases de dados. No diagrama de estrutura de dados é especificado como os dados devem entrar e sair dos processos e serem armazenados no sistema.

Uma ferramenta CASE pode armazenar as descrições em um dicionário/repositório de dados. Completando o diagrama tem-se o modelo lógico do negócio, que é uma representação abstrata do mundo real.

Metodologia Desenvolvimento Ágil

A própria Wikipédia define o desenvolvimento de software ágil, que evoluíram a partir da metade de 1990, como parte de uma reação contra métodos "pesados", caracterizados por uma pesada regulamentação, regimentação e micro gerenciamento usado o modelo em cascata para desenvolvimento. O processo originou-se da visão de que o modelo em cascata era burocrático, lento e contraditório a forma usual com que os engenheiros de software sempre realizaram trabalho com eficiência.

Uma visão que levou ao desenvolvimento de métodos ágeis e iterativos era retorno à prática de desenvolvimento vistas nos primórdios da história do desenvolvimento de software.

Inicialmente, métodos ágeis eram conhecidos como métodos leves. Em 2001, membros proeminentes da comunidade se reuniram em Snowbird e adotaram o nome métodos ágeis. Mais tarde, algumas pessoas formaram A Agile Alliance, uma organização não lucrativa que promove o desenvolvimento ágil.

Os métodos ágeis iniciais - criado a priore em 2000 (ver Manifesto Ágil em <http://agilemanifesto.org/>) - incluíam Scrum(1986), Crystal Clear, Programação extrema (1996), Adaptive Software Development, Feature Driven Development, and Dynamic Systems Development Method (1995).

Metodologia Orientada A Objetos

O desenvolvimento orientado a objetos (OOSD – Object-oriented System Development) encara cada processo como uma coleção de objetos. Os termos encapsulamento, requerimento, herança e classe são básicos dentro do contexto da metodologia. A metodologia possui um conjunto próprio de diagramas e também usa alguns diagramas similares aos da metodologia estruturada. Segundo Rumbaugh a técnica de modelagem de objetos possui quatro fases: análise, projeto do sistema, projeto dos objetos e a implementação.

Nos dias atuais a Metodologia Orientada a Objetos está vencendo a Estruturada. Para facilitar o processo de migração da Estruturada para a Orientada a Objetos, algumas

metodologias “ponte” estão sendo utilizadas na transição. Apesar dessa tendência não podemos nos esquecer da Metodologia Desenvolvimento Ágil que ainda possui muitos adeptos a essa filosofia de desenvolvimento.



Estudo Complementar

http://pt.wikipedia.org/wiki/Metodologia_%28engenharia_de_software%29



Atividades

Você pode se inteirar melhor das Metodologias ouvindo os interessantes podcasts (MP3):

<http://agilcoop.incubadora.fapesp.br/portal/agilcast/episodios/Agilcast01-intro.mp3>

<http://agilcoop.incubadora.fapesp.br/portal/agilcast/episodios/Agilcast07-Perguntas.mp3>



UNIDADE 10

Objetivo: Conceituar de forma geral e ampla da Metodologia Estruturada.

Metodologia Estruturada

Na Análise Clássica de Sistemas é gerado um documento descrevendo o sistema proposto. Esse documento é normalmente chamado de Especificação Funcional do Sistema. Por ser um documento extremamente formal, costumeiramente os usuários assinam e não o leem adequadamente. Praticamente os usuários interpretam como sendo um daqueles contratos bancários, com letras bem pequenas, e que todo mundo assina sem saber muito bem para quê.

As principais desvantagens desse tipo de documento com especificações técnicas clássicas podem ser resumidas da seguinte forma:

- São monolíticos e devem ser lidos do início ao fim;
- São redundantes, dando a mesma informação em diversos locais dentro do documento;
- São difíceis de se modificar e manter. Uma simples mudança nos requisitos do usuário pode acarretar mudanças significativas na especificação;
- São normalmente físicas, ao invés de lógicas, pois descrevem os requisitos em termos do hardware físico, ou do tipo de estrutura física de arquivo que será usado para implementar o sistema. Confundindo-se a discussão sobre o que o usuário efetivamente quer.

No entanto, a Análise Estruturada veio com intuito de quebrar esse paradigma, utilizando de ferramentas gráficas para a documentação, gerando um novo tipo de especificação funcional.

Ferramentas Da Análise Estruturada

As principais ferramentas que a Análise Estruturada criou na época para a devida documentação do Sistema foram:

Diagrama de Fluxo de Dados (DFD)

Modo gráfico de representação da movimentação dos dados num sistema manual, automático ou misto. O DFD deve ser iniciado pelo sistema físico atual, para tanto o entendimento do analista como do usuário. Isso permita que se tenha uma visão top-down da estrutura atual de trabalho dos usuários. Com essa compreensão parte-se para o início do fluxo lógico do sistema a ser proposto.

Dicionário de Dados

É o conjunto organizado das definições lógicas de todos os nomes de dados apresentados no DFD.

Especificação de Processo

Permite que o analista descreva a direção de negócios, representada por cada um dos processos. Devem-se detalhar todos os processos desde aqueles de nível mais baixo representados no DFD. A Especificação de Processo pode ser escrita de várias formas (fórmulas, gráficos), mas normalmente através do Português Estruturado. Esse recurso possui um grupo limitado de verbos e substantivos, organizados a fim de garantir a legibilidade e o rigor.

Diagrama Entidade Relacionamento (DER)

Procura enfatizar os principais objetos ou entidades de dados do Sistema, bem como a relação entre esses objetos. É importante destacar que os DFDs e o DERs enfatizam aspectos diferentes do mesmo Sistema. Logo, existem correspondências um a um que o Analista de Sistemas deve verificar para assegurar um modelo geral coerente.

Características Da Análise Estruturada

Modular

Ela foi estruturada, ao invés de monolítica da Análise Clássica, para o conceito de módulos, quebrando o Sistema em partes lógicas.

Gráfica

Consistindo em mais figuras do que palavras.

Top-Down

Apresentando a descrição do sistema em níveis progressivamente mais detalhados. Começa com uma visão geral para depois entrar nos detalhes.

Lógica

Descreve um modelo independente de implementação.



Estudo Complementar

http://pt.wikipedia.org/wiki/An%C3%A1lise_Estruturada





Atividades

Tente descobrir porque ainda existem muitos sistemas que foram ou são concebidos com a Metodologia Estruturada.



Atividades

Antes de dar continuidades aos seus estudos é fundamental que você acesse sua SALA DE AULA e faça a Atividade 1 no “link” ATIVIDADES.



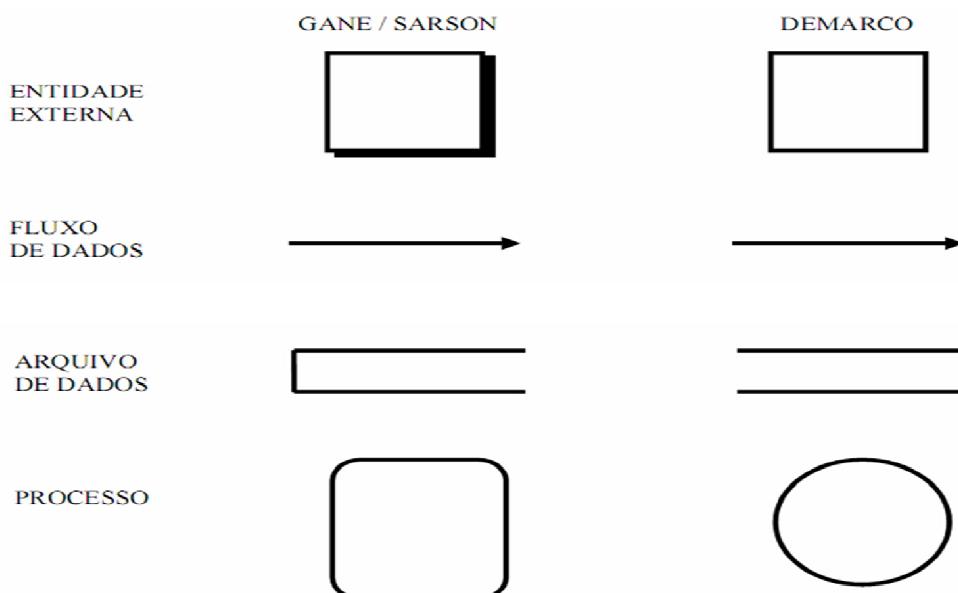
UNIDADE 11

Objetivo: Exemplificar os Diagramas de Fluxo de Dados e os seus principais componentes e tipos.

DFD – Diagrama de Fluxo de Dados

O diagrama de fluxo de dados - DFD - representa o fluxo de dados num sistema de informação, assim como as sucessivas transformações que estes sofrem. O DFD é uma ferramenta gráfica que transcreve, de forma não técnica, a lógica do procedimento do sistema em estudo, sendo usada por diferentes métodos e principalmente pelos classificados como orientados a processos.

O DFD é a ferramenta mais usada para documentar a fase de análise do convencional ciclo de desenvolvimento de sistemas de informação. O diagrama de fluxo de dados apresenta sempre quatro objetos de um sistema de informação: fluxo de dados, processos, arquivos de dados e entidades externas. Esta ferramenta é usada por diferentes autores, por exemplo, Yourdon & DeMarco (veja Figura 11.1) e Gane & Sarson (veja Figura 11.2), que recorrem a métodos e símbolos diferentes para representar cada objeto:



No entanto, qualquer autor que use estes diagramas define os objetos do sistema da mesma forma:

Entidades externas

Pessoa, grupo de pessoas ou subsistema/sistema fora do sistema em estudo que recebem dados do sistema e/ou enviam dados para o sistema. As entidades externas funcionam sempre como origem/destino de dados;

Fluxo de dados

Dados que fluem entre processos, entre processos e arquivos de dados ou ainda entre processos e entidades externas, sem nenhuma especificação temporal (por exemplo ocorrência de processos simultâneos, ou todas as semanas);

Arquivo de dados

Meio de armazenamento de dados para posterior acesso e/ou atualização por um processo;

Processo

Recebe dados de entrada e transforma estes dados num fluxo de saída.

Atribuição De Nomes Aos Objetos

Qualquer objeto do sistema representado no DFD tem de ter um nome elucidativo e claro para que os usuários possam interpretar facilmente o diagrama; os nomes devem refletir exatamente a atividade do sistema.

Todos os autores ditam que o nome de um processo seja constituído por um único verbo e um substantivo, devidamente escolhidos para que transmitam claramente o que o processo faz. Assim verbos como processar, examinar, tratar, nunca devem ser usados, pois são redundantes com o próprio conceito de processo e não deixam claro a própria atividade do processo.

Também, uma vez que o DFD representa logicamente o sistema, abstraindo-se de conceitos físicos, verbos como enviar ou armazenar não devem ser usados, pois têm características físicas.

Certos autores estipulam que o nome atribuído a entidades externas e arquivos de dados deve ser escrito em letras maiúsculas e que o nome atribuído a processos e fluxos de dados deve ser escrito em minúsculas, exceto a primeira letra.

Como Ligar Os Objetos

Os fluxos de dados ligam entre si os outros objetos do sistema representados num DFD (processos, arquivos de dados e entidades externas). Um processo tem, obrigatoriamente, pelo menos um fluxo de entrada e um fluxo de saída, podendo ser a origem de um fluxo para um determinado processo, um arquivo de dados ou uma entidade externa. De igual forma, o destino de um fluxo de um determinado processo pode ser outro processo, um arquivo de dados ou uma entidade externa.

Assim qualquer fluxo de dados tem sempre uma origem e um destino, sendo sempre necessariamente um deles um processo. Um fluxo de dados tem obrigatoriamente um só sentido.

Um arquivo de dados tem também, pelo menos, um fluxo para e/ou um processo (os arquivos de dados estão sempre ligados a processos), não sendo obrigatório ter ambos, pois um arquivo de dados pode só ser atualizado ou só ser acessado pelo sistema em estudo, significando que outro sistema também o utiliza.

Nunca se pode ter num DFD uma ligação entre uma entidade externa e um arquivo de dados, entre dois arquivos de dados e entre duas entidades externas. Neste último caso, se há fluxo entre duas entidades externas ao sistema em estudo, pode-se dizer que esse fluxo não pertence ao referido sistema e assim não deve ser considerado no diagrama.

Elaboração De Um DFD

Embora a prática torne fácil a elaboração de um DFD, é, no entanto, de importância vital efetuar sempre o estudo cuidadoso da definição da fronteira que delimita o sistema, pois só a partir daí é possível identificar os elementos que vão fazer parte do diagrama.

Para a elaboração de um DFD utiliza-se a abordagem “top-down” em que cada um dos diferentes níveis de detalhe do sistema em estudo é mostrado através de diferentes níveis de DFD. A primeira representação do sistema é elaborada através de um diagrama conhecido como diagrama de contexto. Este diagrama, denominado nível 0, é representado através de um processo e dos fluxos de entrada e saída do sistema, o que permite delimitar a área em estudo.

Depois, cada processo de DFD de nível 1 pode ser decomposto sucessivamente noutros DFDs onde já se mostram mais detalhes da lógica de procedimento. Esta técnica de subdividir DFDs de nível superior em DFDs que representam sucessivamente o sistema com mais detalhe é conhecida por “levelling”.

Não há uma regra geral que diga quando se deve acabar com esta subdivisão; alguns autores defendem que é quando os processos estão sob a forma de primitiva funcional, outros que não se devem ultrapassar sete níveis de detalhe. No entanto, todos os autores dizem que quando se decompõe um processo num outro DFD de detalhe deve haver conservação de fluxos, isto é, os fluxos que entram e saem do processo do DFD de nível superior, têm também que entrar e sair no DFD que representa a decomposição desse processo; esta propriedade é denominada por “balancing”.

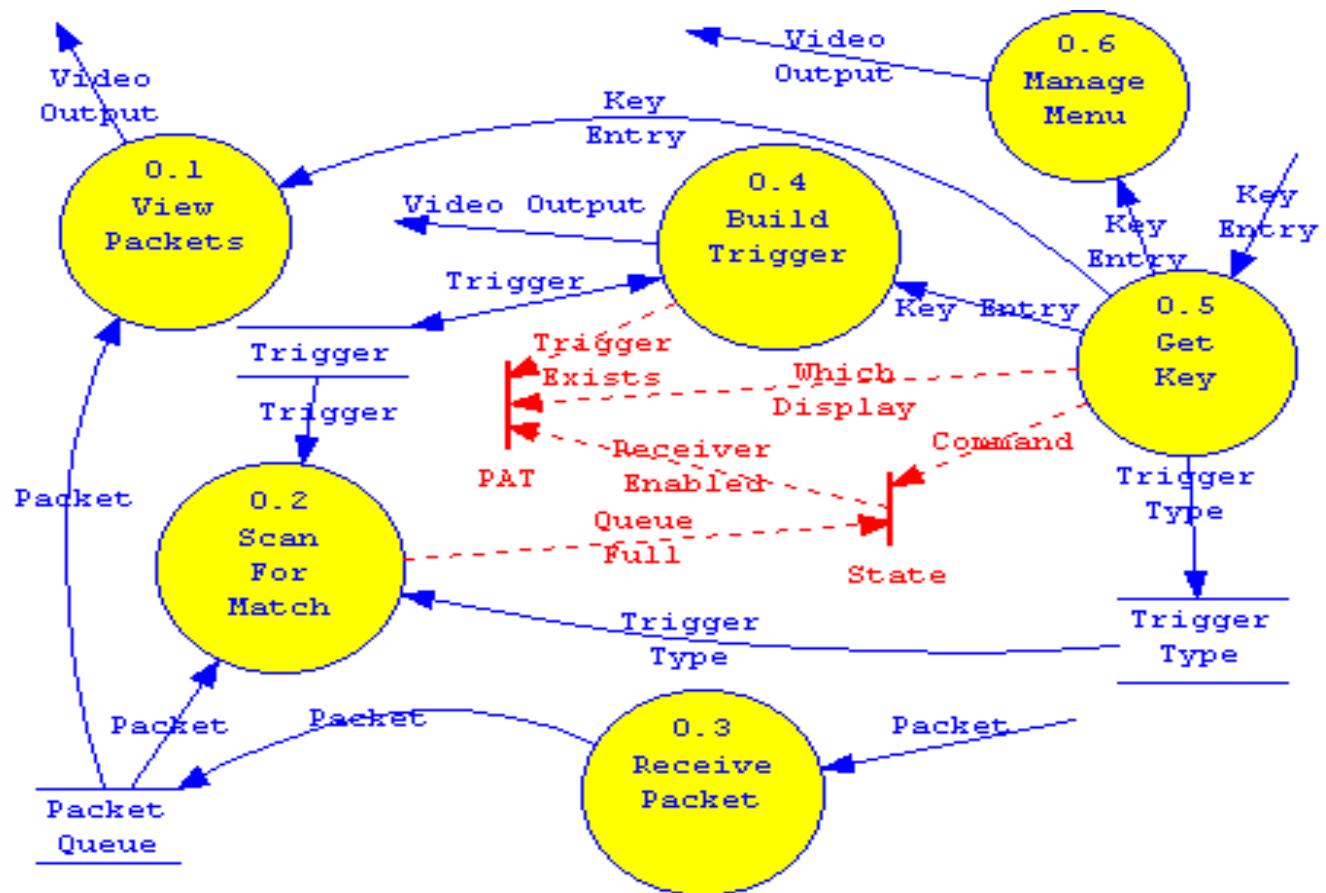


Figura 11.2 - DFD segundo Yourdon & DeMarco

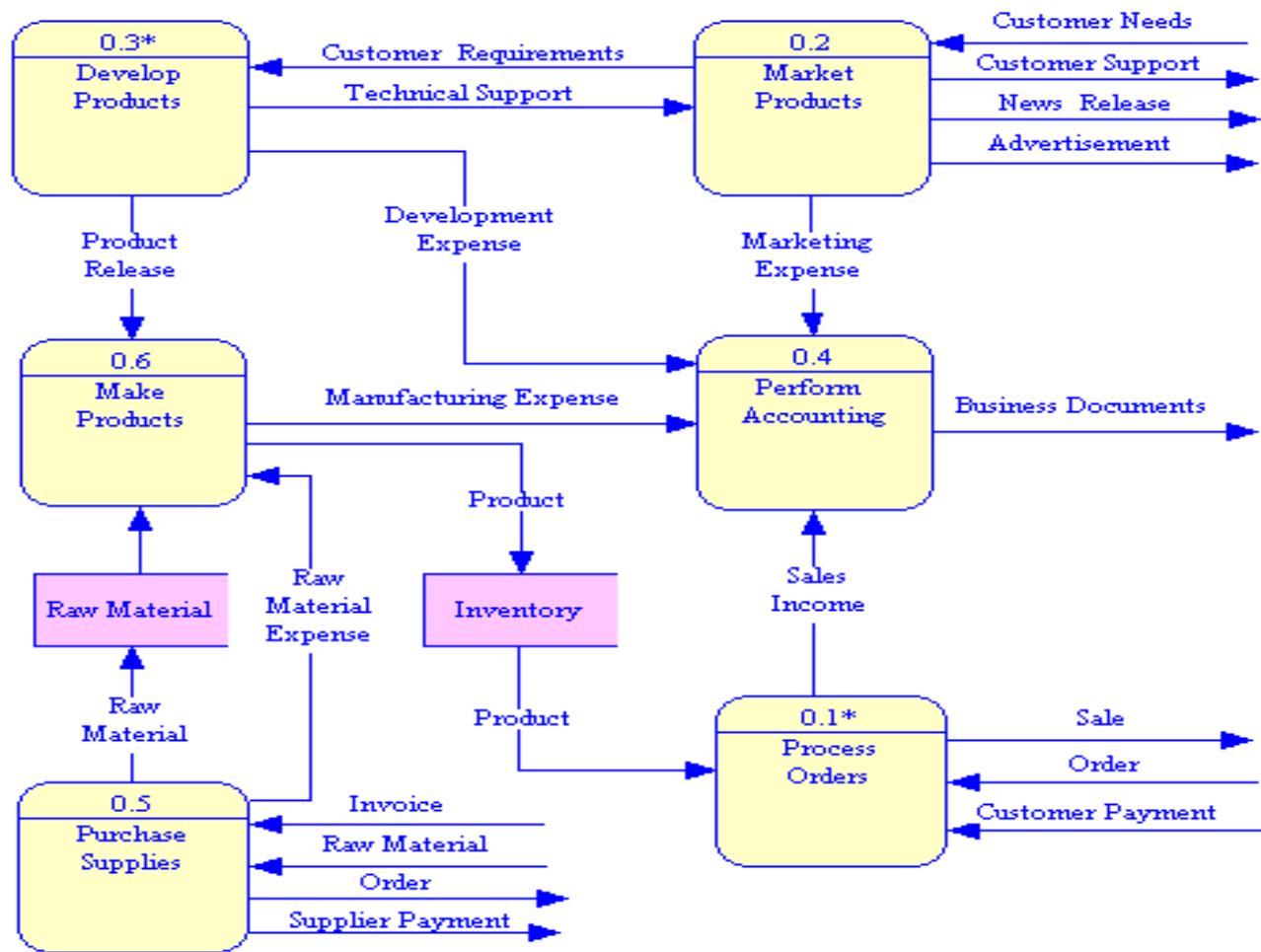


Figura 11.2 - DFD segundo Gane & Sarson



Estudo Complementar

<http://pt.wikipedia.org/wiki/DFD>



Atividades

Com base nas figuras 11.1 e 11.2, elabore uma relação das principais diferenças entre os DFDs dos autores Yourdon & DeMarco e Gane & Sarson



UNIDADE 12

Objetivo: Conceituar e diferenciar o Modelo e o Diagrama de Entidades e Relacionamentos.

MER e DER – Modelo e Diagrama de Entidades e Relacionamentos

O Modelo Conceitual de Dados (ou Modelo de Entidades e Relacionamentos – MER) é representado por um gráfico denominado DIAGRAMA de ENTIDADES e RELACIONAMENTOS (DER), proposto por Peter Chen (1976). Trata-se de um diagrama que detalha as associações existentes entre as entidades de dados e utiliza componentes semânticos próprios.

A Elaboração Do Modelo De Entidades E Relacionamentos Segue Aos Seguintes Princípios:

- Cada Entidade é representada por um retângulo na posição horizontal;
- Cada tipo de relacionamento é representado por um losango;
- Pode haver um tipo de relacionamento entre mais do que duas Entidades;
- Os relacionamentos podem conter atributos;
- Pode haver mais de um tipo de relacionamento entre duas Entidades.

Para a compreensão do Modelo E x R é necessário considerar cada Entidade sob o enfoque de dados, não levando em conta os processos (procedimentos ou rotinas). Os relacionamentos se dão entre os dados de uma Entidade em relação aos dados das demais Entidades que formam o Modelo.

O princípio básico é o da Teoria de Conjuntos. Cada Entidade Conceitual deve ser vista como sendo um conjunto que pode ou não ter relacionamento (interseção) com outro conjunto.

Resumidamente, é a seguinte terminologia básica do MER:

ENTIDADE

São os componentes físicos e abstratos utilizados pela empresa, sobre os quais são armazenados dados;

ATRIBUTO

Corresponde à representação de propriedades de uma Entidade. Um atributo não tem vida própria ou independente. Existe apenas para caracterizar uma Entidade;

OCORRÊNCIA

Conjunto de atributos de uma determinada Entidade;

RELACIONAMENTO

É uma correspondência entre duas entidades. Uma associação entre dois conjuntos de dados (entidades);

IDENTIFICADOR ou ATRIBUTO DETERMINANTE

Um atributo ou uma coleção de atributos que determina de modo único uma Ocorrência de Entidade;

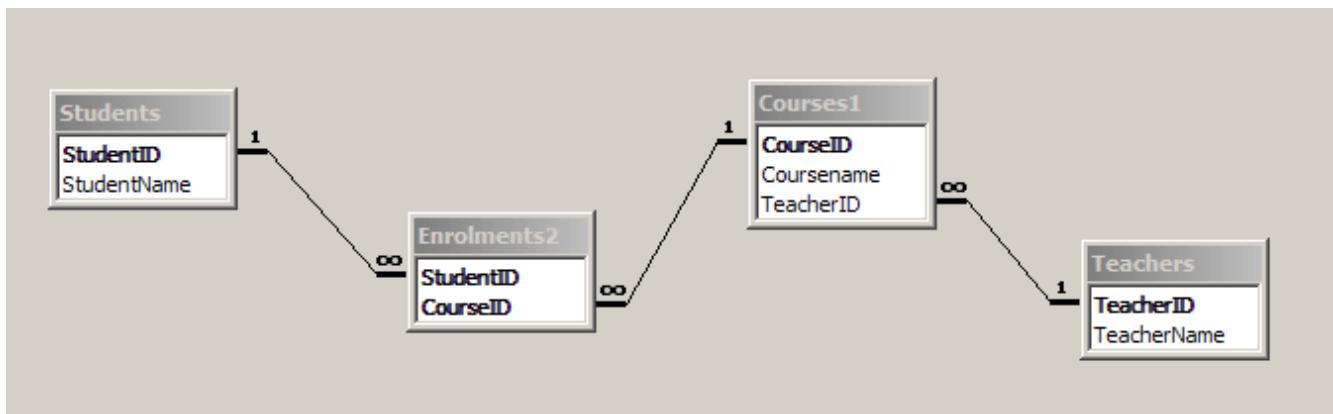
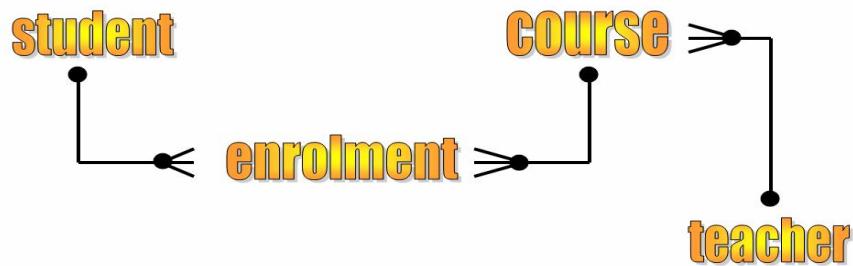
GRAU DE RELACIONAMENTO

O número de entidades que participam da associação;

CLASSE DE RELACIONAMENTO ou CARDINALIDADE

Quantas ocorrências de cada entidade estão envolvidas no relacionamento. Pode ser:

- **1:1** (um para um)
- **1:n** (um para muitos)
- **m:n** (muitos para muitos)



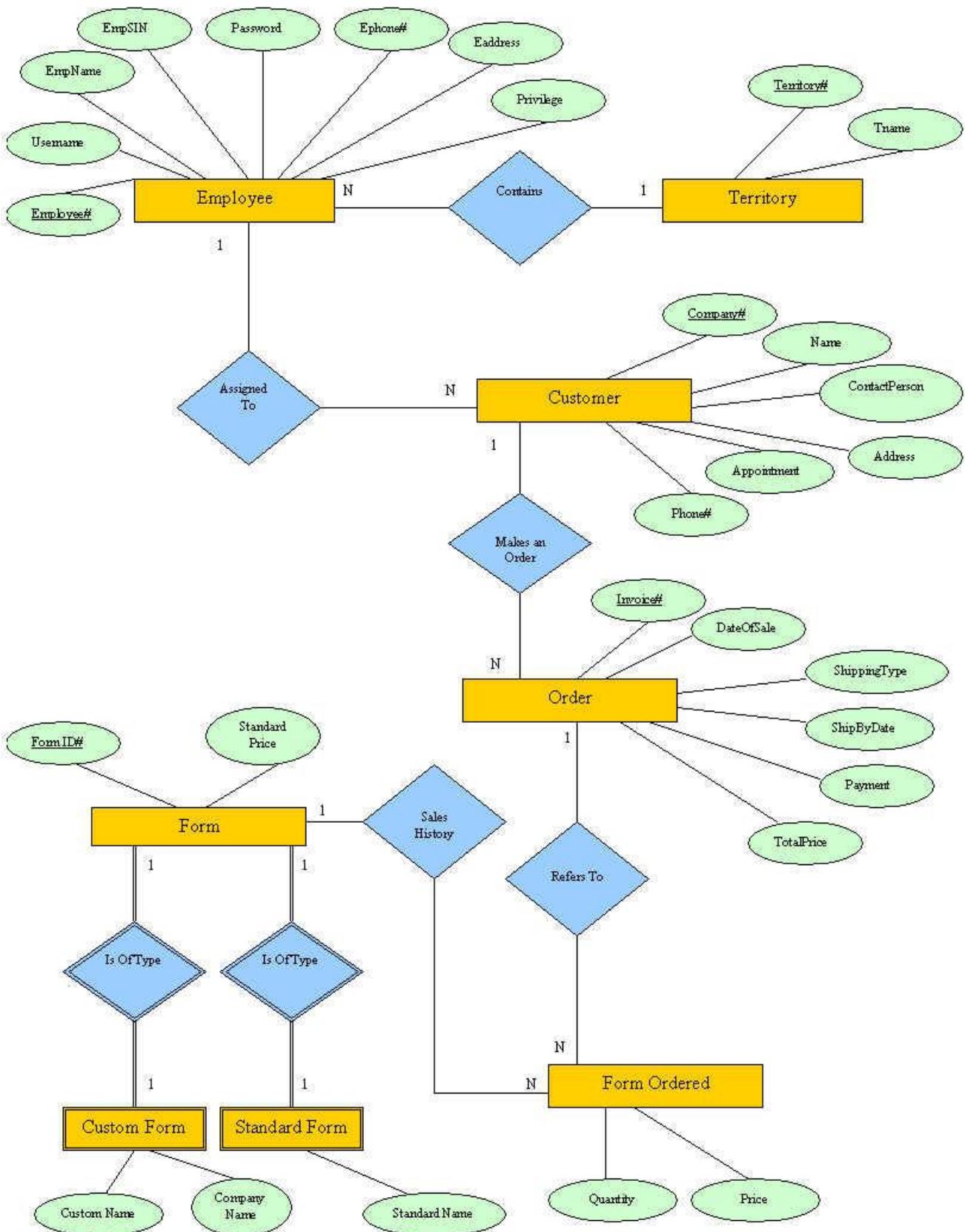
Criação Do Diagrama E-R (DER)

1. A descoberta dos relacionamentos existentes entre as ENTIDADES é realizada pelo analista e usuário, obedecendo-se as seguintes etapas:
2. Individualizar uma ocorrência de uma entidade que integra o modelo conceitual;
3. Questionar se há algum relacionamento entre a ocorrência da entidade de origem com outra entidade do modelo (entidade de destino);
4. No caso de existir o relacionamento, batizá-lo com um nome representativo da associação;
5. Assinalar a classe de relacionamento entre a entidade origem e a entidade destino;

6. Questionar qual a classe de relacionamento entre a entidade destino e sua entidade de origem;
7. Assinalar a classe de relacionamento entre a entidade destino e a sua entidade de origem;
8. Realizar as etapas de 1 a 6 até que todos os relacionamentos do modelo sejam analisados, classificados os seus graus e assinalados no modelo.

Recomendações Para Criação Do Diagrama E-R (DER)

1. Antes de começar a modelar, conheça o “mundo real”.
2. Identifique quais são as ENTIDADES.
3. Para cada Entidade represente seus ATRIBUTOS.
4. Confronte cada Entidade consigo mesma e com as demais na procura de possíveis RELACIONAMENTOS
5. Verifique a existência de ATRIBUTOS de RELACIONAMENTO.
6. Para relacionamentos múltiplos estude a necessidade de AGREGAÇÕES.
7. Desenhe o DER, com todas as Entidades, Atributos, Relacionamentos, Classes e Restrições de Totalidade.
8. Analise cuidadosamente todas as restrições que você impôs.
9. Até que você e os seus usuários estejam convencidos de que o DER reflete fielmente o “mundo real”, volte ao item 1.





Estudo Complementar

http://pt.wikipedia.org/wiki/Modelo_de_Entidades_e_Relacionamentos

http://pt.wikipedia.org/wiki/Diagrama_entidade_relacionamento



Atividades

Com base na última imagem apresentada nesta unidade faça uma análise crítica sobre o respeito às recomendações para a criação do Diagrama E-R (DER).



UNIDADE 13

Objetivo: Apresentar a evolução natural da Metodologia Estruturada ao longo do tempo.

Evolução da Metodologia Estruturada

A Análise Estruturada foi popularizada por Tom DeMarco, surgindo então, um importante paradigma de desenvolvimento de sistemas. Obras importantes tais como Gane & Sarson, Yourdon, Constantine e Page-Jones nas áreas de análise e projeto estruturados foram decisivas na aprendizagem desse paradigma.

Passamos a conviver com Diagramas de Fluxos de Dados (DFD), Diagramas de Entidades e Relacionamentos (DER), entre outros como vimos nas unidades anteriores. Muito se conseguiu melhorar no processo de desenvolvimento e se difundir com esta metodologia, fazendo com que o desenvolvimento de sistemas fosse enxergado com mais esmero, respeitando sua complexidade. Tudo isso levou consecutivamente a produtos de melhor qualidade.

Hoje possuímos ótimos softwares que, tendo sido desenvolvidos plenamente dentro da modelagem de análise estruturada ou essencial, conseguiram obter seu “lugar ao sol”.

Todavia, existem problemas nessa metodologia. Há uma dificuldade em garantir compatibilidade entre as fases de análise e projeto e, posteriormente, do projeto para a implementação. Na grande maioria das vezes, grandes alterações ou extensões dos modelos criados geram um grande esforço.

Não podemos esquecer que a comunicação entre desenvolvedores e usuários é difícil, em virtude dos diagramas não serem muito expressivos fora da compreensão da equipe de desenvolvimento. Validações dos usuários até ocorrem, mas no geral não refletem o universo dos requisitos do sistema.

É fácil lembrar os rostos dos usuários diante dos cuidadosamente desenhados DFD's e DER's. Até os desenvolvedores mais resistentes à mudança de paradigma reconhecem que não é fácil conseguir uma validação precisa de um usuário, a partir das ferramentas usadas na análise estruturada.

Assim, o resultado final é imprevisível. Além de tudo temos o problema mais delicado: processos e dados são vistos de forma independente.

Talvez esses problemas não fossem tão graves há 20 anos, época em que os sistemas eram menores, possuíam menos complexidades, a Internet ainda não existia em nossas vidas e os usuários não tinham noção do quanto podiam pedir!

Mas, atualmente, a complexidade, a urgência e a adaptabilidade dos novos aplicativos têm levado a se repensar os prós e contras dessa abordagem. Estamos vivendo uma era na qual precisamos mais do que entregar softwares no prazo, dentro do orçamento e sem falhas – até porque isto é obrigação do desenvolvedor.

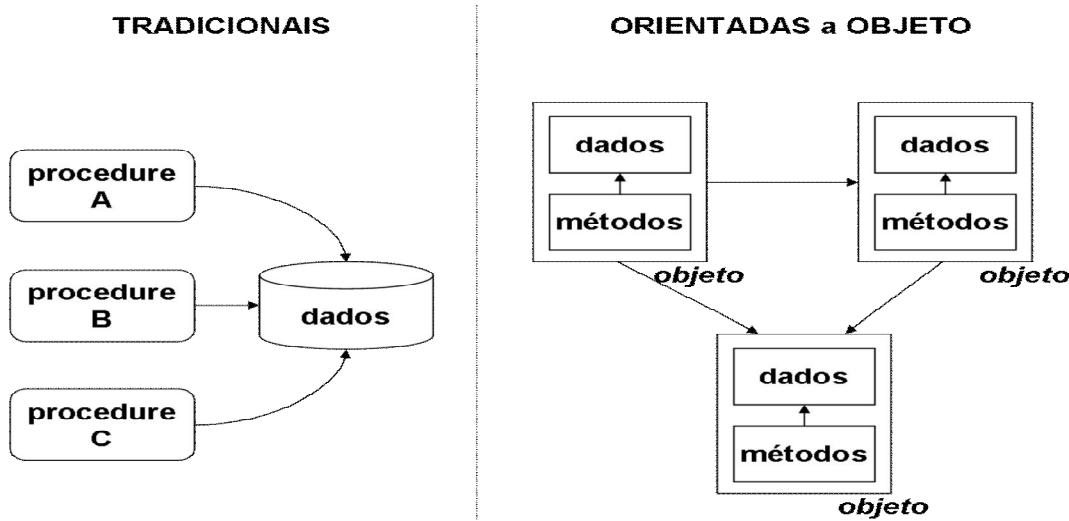
Precisamos desenvolver novos softwares gastando menos tempo, economizando efetivamente e oferecendo um algo mais para o usuário. Lembra-se do comercial que dizia: “Não basta ser pai, é preciso participar”? E é isso que precisamos fazer: não basta entregar um bom software, precisamos emocionar nossos usuários.

A partir de meados da década de 70, métodos orientados a objetos começaram a surgir, vislumbrando a mudança de paradigma. A programação orientada a objetos já estava amadurecendo, bastava, portanto, criar para ela uma metodologia – e começamos a conhecer a análise orientada a objetos.

O principal ganho com esse novo paradigma é o fato de que se uniformizaram os modelos usados para análise, projeto e implementação. Os principais diagramas são usados em todas as fases, mudando-se apenas sua visão.

Partindo do conceito de orientação de objetos, há a unificação da perspectiva funcional e de dados. Não podemos esquecer o melhor dos mundos: a comunicação entre desenvolvedores e usuários tornou-se mais facilitada.

Os usuários conseguem participar muito mais ativamente do processo de desenvolvimento, pela análise e validação dos diagramas apresentados. Assim garante-se que praticamente não existirão surpresas quanto à compreensão de requisitos, na entrega final do produto.



As linguagens tradicionais possuem o foco nos programas, responsáveis pela parte ativa da aplicação. Os dados são manuseados de forma passiva, perdendo, em alguns casos, sua importância de contexto.

Em contrapartida, nas linguagens orientadas a objeto, os dados estão protegidos por uma cápsula, na qual residem procedimentos que intermediam o acesso a eles. Eles passam a ser as estrelas do espetáculo.

A Orientação a Objetos veio propiciar à comunidade de desenvolvedores: aumento de produtividade, diminuição do custo de desenvolvimento e principalmente de manutenção, maior portabilidade e reutilização de código.



Estudo Complementar

http://www.training.com.br/lpmaia/pub_prog_oo.htm

<http://www.ccuec.unicamp.br/revista/infotec/artigos/leite Rahal.html>



Atividades

Com base no material desta unidade você consegue identificar as diferenças básicas entre as análises tradicionais e a análise orientada a objetos? Você acha que era necessário aplicar os conceitos de orientação a objetos no passado? Justifique a resposta.



UNIDADE 14

Objetivo: Explicar o conceito de modelagem pela visão orientada a objetos.

Modelagem de Sistemas Orientados a Objetos

A modelagem é a parte central de todas as atividades que levam à implantação de um bom software. Construímos modelos para:

- **Comunicar** a estrutura e o comportamento desejados do sistema.
- **Visualizar** e controlar a arquitetura do sistema.
- **Compreender** melhor o sistema que estamos elaborando.
- **Expor** mais claramente oportunidades de simplificação e reaproveitamento do sistema em estudo.
- **Gerenciar** os riscos.

No próprio livro dos autores do UML existe um exemplo muito bacana de modelagem. Eles começam com a situação da construção de uma casa para um cachorro. Por ser uma construção para um animal, que no final pouco irá reclamar dos detalhes, pois o cachorro gosta mais do dono do que da casa, os cuidados a serem tomados são mínimos. Pegam-se tábuas, pregos e algumas ferramentas básicas como martelo, serrate e metro e manda-se ver ... Bem, mesmo que não venha a ser a contento do cão, no pior caso, se ele não gostar da nova casa, pode-se ainda trocar de animal.

Em seguida, os autores evoluem no exemplo simulando que agora a casa será construída para a família. Os cuidados já serão outros, concorda?

No mínimo, para se ter uma casa de qualidade, exigirá algum esboço do projeto, para que todos possam contribuir e visualizar melhor o que será construído. E com a finalidade também de ver detalhes práticos de energia, circulação e encanamento. A partir desses

planos, você poderá começar a fazer uma estimativa razoável da quantidade de tempo e de material necessários para essa tarefa.

Embora seja humanamente possível construir uma casa sozinho, você logo descobrirá que será muito mais eficiente trabalhar com outras pessoas, talvez terceirizando vários serviços básicos ou comprando material pré-fabricado. Desde que você se mantenha fiel aos planos e permaneça dentro dos limites de tempo e custos, provavelmente sua família ficará bem satisfeita.

Se não der certo, a melhor solução não será trocar de família. Portanto, será melhor definir as expectativas desde o início e gerenciar qualquer modificação com muita cautela.

Os autores terminam o exemplo com o cenário de uma construção de um prédio comercial. Para construir um prédio comercial com vários andares, não será uma boa ideia começar com uma pilha de tábuas, alguns pregos e algumas ferramentas básicas. Como provavelmente você usará o dinheiro de outras pessoas, como os diretores da empresa, eles exigirão saber o tamanho, a forma e o estilo do futuro prédio.

Muitas vezes, essas pessoas mudarão de ideia, mesmo depois de iniciada a construção. Valerá a pena fazer um planejamento rigoroso, pois os custos de qualquer erro serão altos. Você será apenas uma parte de um grupo bem maior, responsável pelo desenvolvimento e pela entrega do prédio. Assim, a equipe precisará de todos os modelos e esboços do projeto para poderem se comunicar entre si, e para futuras manutenções.

Desde que você consiga as pessoas certas e as ferramentas adequadas, além de gerenciar de maneira ativa o processo de transformação de um conceito de arquitetura em realidade, provavelmente acabará obtendo um prédio que satisfará seus futuros ocupantes. Caso pretenda continuar construindo prédios, você tentará encontrar um equilíbrio entre os desejos dos futuros ocupantes e a realidade das tecnologias de construção, além de manter um relacionamento profissional com os demais integrantes de sua equipe, nunca os colocando em risco, nem exigindo tanto que eles acabem exaustos ou doentes.

Curiosamente, muitas empresas de desenvolvimento de software começam querendo construir prédios de relativa complexidade, como se estivessem fazendo uma casinha de cachorro.

Se você realmente quiser construir softwares equivalentes a uma casa ou a um prédio de qualidade, o problema não se restringirá a uma questão de escrever uma grande quantidade de software – de fato, o segredo estará em criar o código correto e pensar em como será possível elaborar menos software.

Isso faz com que o desenvolvimento de software de qualidade se torne uma questão de arquitetura, processo e ferramentas.

Ainda assim, muitos projetos são iniciados parecendo uma casa de cachorro, mas crescem com a grandeza de um prédio simplesmente porque são vítimas de seu próprio sucesso. Chega um momento em que, caso não tenham sido consideradas questões referentes à arquitetura, a processos e a ferramentas, a casa de cachorro, agora ampliada para um grande prédio, sucumbirá ao seu próprio peso. Qualquer erro na casa de cachorro poderá deixar seu cão insatisfeito. A falha na construção de um grande prédio afetará materialmente e fisicamente seus ocupantes e moralmente a toda equipe de trabalho.

Existem muitos elementos que contribuem para uma empresa de software de sucesso. Mas com certeza um desses vitais componentes é a utilização efetiva da modelagem.



Estudo Complementar

http://imasters.uol.com.br/artigo/3069/uml/como_modelar/



Atividades

Avalie na sua atual empresa, ou em empresa de colegas, qual a forma com que é encarado o processo de modelagem de sistemas. Eles ainda estão no nível de arquitetos de casinha de cachorro, ou já chegaram ao nível de grandes construtores? Quais seriam as principais ações a serem implementadas para corrigir isso? Ou quais são os pontos fortes e fracos da situação atual?



UNIDADE 15

Objetivo: Definir de forma prática o que vem a ser Modelo.

O que é Modelo?

Afinal o que é modelo? Falando de uma maneira bem simples:

Um modelo é uma simplificação da realidade.

Os modelos podem ser estruturais, dando ênfase à organização do sistema, ou podem ser comportamentais, dando ênfase à dinâmica do sistema. Por que fazer a modelagem? Existe um motivo fundamental:

Construímos modelos para compreendemos melhor o sistema que estamos desenvolvendo.

Com a modelagem, alcançamos quatro objetivos:

1. *Ajudam a visualizar o sistema como ele é ou como desejamos que seja.*
2. *Permitem especificar a estrutura ou o comportamento de um sistema.*
3. *Proporcionam um guia para a construção do sistema.*
4. *Documentam as decisões tomadas.*

A modelagem não se restringe a grandes sistemas. Até os softwares equivalentes a “casas de cachorro” poderão receber os benefícios da modelagem. Porém, é absolutamente verdadeiro que, quanto maior e mais complexo for o sistema, maior será a importância da modelagem, por uma razão bem simples:

Construímos modelos de sistemas complexos porque não é possível comprehendê-los em toda a sua totalidade.

Existem limites para a capacidade humana de compreender complexidades. Com a ajuda da modelagem, delimitamos o problema que estamos estudando, restringindo nosso foco a um único aspecto por vez.

Em essência esse é o procedimento de “dividir para conquistar”, do qual o matemático e filosófico René Descartes falava em seu “Discurso sobre o Método” há anos: ataque um problema difícil, dividindo-o em vários problemas menores que você pode solucionar.

Além disso, com o auxílio da modelagem, somos capazes de ampliar o intelecto humano. Um modelo escolhido de maneira adequada permitirá a quem usa a modelagem trabalhar em níveis mais altos de abstração.

Qualquer projeto será beneficiado pelo uso de algum tipo de modelagem. Inclusive no setor de softwares comerciais, em que às vezes é mais comum distribuir softwares inadequados devido à produtividade oferecida pelas linguagens de programação visual, a modelagem poderá auxiliar a equipe de desenvolvimento a visualizar melhor o planejamento do sistema e permitir que o desenvolvimento seja mais rápido.

Quanto mais complexo for o sistema, maior será a probabilidade de ocorrência de erros ou de construção de itens errados, caso não haja qualquer modelagem. Todos os sistemas úteis e interessantes apresentam uma tendência natural para se transformarem em algo mais complexo ao longo do tempo. Portanto, ainda que considere não ser preciso fazer a modelagem hoje, à medida que o seu sistema evoluir, você se arrependerá dessa decisão, e aí poderá ser tarde demais.

São várias as razões da utilização de modelos para a construção de sistemas. Abaixo são enumeradas algumas dessas razões:

Analogias Com Outras Engenharias

Na construção civil, frequentemente há profissionais analisando as plantas da construção sendo realizada. A partir dessas, que podem ser vistas como modelos, os homens tomam decisões sobre o andamento da obra. Modelos de sistemas de software não são muito

diferentes dos modelos da engenharia civil. E nós temos outros exemplos de modelos semelhantes tais como maquetes de edifícios e de aviões, e plantas de circuitos eletrônicos.

Gerenciamento da Complexidade

Uma das principais razões pelas quais modelos são utilizados é que há limitações no ser humano em lidar com a complexidade. Pode haver diversos modelos de um mesmo sistema, cada modelo descrevendo uma perspectiva do sistema a ser construído. Por exemplo, um avião pode ter um modelo para representar sua parte elétrica, outro modelo para representar sua parte aerodinâmica etc. Através de modelos de um sistema, os indivíduos envolvidos no seu desenvolvimento podem fazer e predizer comportamentos do sistema. Como cada modelo representa uma perspectiva do sistema, detalhes irrelevantes que podem dificultar o entendimento do sistema podem ser ignorados por um momento estudando-se separadamente cada um dos modelos.

Comunicação Entre As Pessoas Envolvidas

Certamente, o desenvolvimento de um sistema envolve a execução de uma quantidade significativa de atividades. Essas atividades se traduzem em informações sobre o sistema em desenvolvimento. Grande parte dessas informações corresponde aos modelos criados para representar o sistema. Nesse sentido, os modelos de um sistema servem também para promover a difusão de informações relativas ao sistema entre os indivíduos envolvidos em sua construção. Além disso, diferentes expectativas em relação ao sistema geralmente aparecem quando da construção dos seus modelos, já que estes servem como um ponto de referência comum.

Redução Dos Custos No Desenvolvimento

No desenvolvimento de sistemas, seres humanos estão invariavelmente sujeitos a cometerem erros, que podem se tanto erros individuais quanto erros de comunicação entre os membros da equipe. Certamente, a correção desses erros é menos custosa quando detectada e realizada ainda nos modelos do sistema (por exemplo: é muito mais fácil corrigir uma maquete do que pôr abaixo uma parte de um edifício).

Lembre-se: Modelos de sistemas são mais baratos de construir do que sistemas.

Consequentemente, erros identificados em modelos têm um impacto menos desastroso nos sistemas.

Predição Do Comportamento Futuro Do Sistema

O comportamento do sistema pode ser discutido através da análise dos seus modelos. Os modelos servem como um grande laboratório, onde diferentes soluções para um problema relacionado à construção do sistema podem ser experimentadas.

Nas próximas unidades apresentaremos diversos modelos cujos componentes são desenhos gráficos que seguem algum padrão lógico. Esses desenhos são normalmente denominados de diagramas. Um diagrama é uma apresentação de uma coleção de elementos gráficos que possuem um significado predefinido. Dado um modelo de uma das perspectivas de um sistema, diz-se que o seu diagrama, juntamente com a informação textual associada, forma a documentação desse modelo.



Estudo Complementar

Modelo de Definição de Sistemas da Microsoft:

<http://www.microsoft.com/portugal/windowsserversystem/dsi/sdm.mspx>

http://pt.wikipedia.org/wiki/Modelagem_computacional





Atividades

Em sua visão qual é a importância da criação de modelos no processo de desenvolvimento de sistemas? Justifique.



UNIDADE 16

Objetivo: Apresentar formalmente os quatro princípios da modelagem.

Princípios da Modelagem

O uso da modelagem tem uma rica história em todas as disciplinas de engenharia e arquitetura. Essa experiência sugere quatro princípios básicos de modelagem.

PRIMEIRO PRINCÍPIO

A escolha do modelo tem profunda influência sobre a maneira como um determinado problema é atacado e como uma solução é definida.

Em outras palavras, escolha bem os seus modelos. Os modelos corretos iluminarão de modo brilhante os problemas de desenvolvimento mais complicados, proporcionando conclusões que simplesmente não seriam possíveis de outra maneira; modelos inadequados causarão confusões, desviando a atenção para questões irrelevantes.

Deixando de lado o desenvolvimento de software por um instante, suponha que você esteja tentando solucionar um problema de física quântica. Certos problemas, como a interação de fótons no tempo-espacô, implicam uma matemática maravilhosamente complexa. Escolha um modelo que não seja o de cálculo e imediatamente essa complexidade inerente se tornará manipulável.

De modo semelhante, em um domínio inteiramente diferente, suponha que você está construindo um novo prédio e está interessado em saber como ele se comportará quando houver fortes ventanias. Construindo um modelo físico e submetendo-o a testes de túneis de vento, você aprenderá algumas coisas interessantes, apesar de os materiais em escalas menores não se flexionarem exatamente como em escalas maiores. Assim, se elaborar um modelo matemático e depois submetê-lo a simulações, você aprenderá algumas coisas

diferentes e provavelmente também será capaz de trabalhar com um número maior de novos cenários do que se estivesse utilizando modelos físicos. A partir de testes contínuos e rigorosos com seus modelos, você acabará obtendo um nível de confiança muito superior em relação ao fato de que o sistema, cuja modelagem foi realizada, de fato se comportará da maneira esperada no mundo real.

Em relação aos softwares, a escolha de modelos poderá ser modificada, de maneira significativa, de acordo com sua visão de mundo. Construindo um sistema a partir da perspectiva de um desenvolvedor de bancos de dados, provavelmente você atribuirá o foco a modelos de relacionamentos entre entidades, cujo comportamento tende a privilegiar procedimentos armazenados e os eventos que os iniciam. Construindo um sistema a partir da perspectiva de um analista de análise estruturada, provavelmente usará modelos centrados em algoritmos, com o respectivo fluxo de dados de um processo para outro.

Construindo um sistema a partir da perspectiva de um desenvolvedor orientado a objetos, provavelmente trabalhará com um sistema cuja arquitetura estará centrada em várias classes e os padrões de interação que determinarão como essas classes funcionarão em conjunto.

Qualquer uma dessas soluções poderá ser correta para uma determinada aplicação e cultura de desenvolvimento, apesar de a experiência indicar que a perspectiva orientada a objetos é superior para a criação de arquiteturas flexíveis, inclusive no caso de sistemas que poderão conter grandes bancos de dados ou vários componentes computacionais. Apesar desse fato, o ponto mais importante é que cada visão de mundo conduz a um tipo diferente de sistema, com custos e benefícios diversos.

SEGUNDO PRINCÍPIO

Cada modelo poderá ser expresso em diferentes níveis de precisão.

Ao desenvolver um sistema complexo, às vezes poderá ser necessária uma visão panorâmica – por exemplo, para ajudar os investidores a visualizar a aparência e o funcionamento do futuro sistema. Em outras situações, poderá ser preciso retornar ao nível

dos alicerces – por exemplo, quando existe uma rotina cuja execução é crítica ou um componente estrutural pouco comum.

O mesmo é verdadeiro em relação aos modelos de software. Às vezes, um modelo da interface para o usuário, de execução rápida e simples, será exatamente o que você precisará; em outros casos, será necessário retornar a níveis mais baixos, como ao especificar interfaces para várias plataformas ou ao enfrentar congestionamentos em uma rede. Em qualquer situação, os melhores tipos de modelos serão aqueles que permitem a escolha do grau de detalhamento, dependendo de quem esteja fazendo a visualização e por que deseja fazê-la. Um analista ou um usuário final dirigirá a atenção em direção a questões referentes ao que será visualizado; o desenvolvedor moverá o foco para a maneira como esses objetos funcionarão. Todos esses observadores desejarião visualizar o sistema em níveis diferentes de detalhamento em situações distintas.

TERCEIRO PRINCÍPIO

Os melhores modelos estão relacionados à realidade.

O modelo físico de um prédio, que não responda da mesma forma que os materiais reais, terá um valor apenas limitado; o modelo matemático de um avião, em que são consideradas apenas condições de vôo ideais e fabricação perfeita, poderá ocultar características potencialmente fatais do avião de verdade. Será melhor utilizar modelos que tenham uma clara conexão com a realidade e, nos casos em que essa conexão seja fraca, saber, de maneira exata, como esses modelos diferem do mundo real. Todos os modelos simplificam a realidade; o segredo será ter certeza de que sua simplificação não ocultará detalhes importantes.

No caso dos softwares, o tendão de Aquiles das técnicas de análise estruturada está no fato de não haver uma conexão básica entre o modelo de análise e o modelo de projeto do sistema. Falhando a ponte sobre essa fenda, ao longo do tempo aparecerá uma divergência entre o sistema concebido e o sistema construído. Nos sistemas orientados a objetos, é

possível estabelecer alguma conexão entre todos os pontos de vista quase independentes de um sistema em uma mesma totalidade semântica.

QUARTO PRINCÍPIO

Nenhum modelo único é suficiente. Qualquer sistema, não trivial, será melhor investigado por meio de um pequeno conjunto de modelos, quase independentes um do outro.

Para a construção de um prédio, não existe um único conjunto de esboços de projetos capaz de revelar todos os detalhes da construção. Pelo menos, serão necessárias plantas baixas, aéreas, elétricas, de circulação e de água e esgoto.

A expressão funcional aqui utilizada é “quase independente”. Nesse contexto, a expressão significa modelos que possam ser criados e estudados separadamente, mas que continuam inter-relacionados. Assim como no caso de um prédio, você poderá estudar apenas as plantas relacionadas à energia elétrica, mas também poderá ver o respectivo mapa na planta baixa e talvez até sua interação com o direcionamento de canos na planta de água e esgoto.

O mesmo é verdadeiro em relação a sistemas de software orientados a objetos. Para compreender a arquitetura desses sistemas, você precisará recorrer a várias visões complementares e inter-relacionadas: a visão dos casos de uso (expondo os requisitos do sistema), a visão de projeto (captando o vocabulário do espaço do problema e do espaço da solução), a visão do processo (modelando a distribuição dos processos e threads do sistema), a visão da implementação do sistema (direcionando a realização física do sistema) e a visão da implantação (com o foco voltado para questões de engenharia de sistemas). Cada uma dessas visões poderá conter aspectos estruturais como também aspectos comportamentais. Em conjunto, essas visões representam a base do projeto do software.

Dependendo da natureza do sistema, alguns modelos poderão ser mais importantes do que outros. Por exemplo, no caso de sistemas que utilizam muitos dados, predominarão modelos voltados para a visão estática do projeto. Em sistemas centrados no uso de GUI, são importantes as visões estáticas e dinâmicas dos casos de uso. Em sistemas de execução

crítica em tempo real, a visão dinâmica do processo tende a ser mais importante. Por fim, em sistemas distribuídos, como aqueles encontrados em aplicações que utilizam a Web, os modelos de implementação e de implantação são os mais importantes.



Estudo Complementar

<http://www.ppgia.pucpr.br/~alcides/Teaching/ProgramasAprendizagem/ModelagemOrientadaObjetos/Introducao.html>



Atividades

Como você aplicaria os quatro princípios da modelagem ao seu dia a dia de Análise de Sistemas? A modelagem altera as visões que temos de um sistema?



UNIDADE 17

Objetivo: Definir e apresentar a linguagem de modelagem unificada.

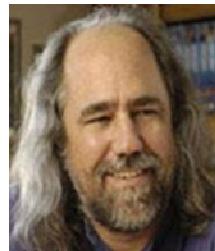
UML – Unified Modeling Language

Pelas próprias palavras dos criadores Booch, Rumbaugh e Jacobson, chamados carinhosamente de “os três amigos”, eles definem a UML da seguinte forma:

“A UML proporciona uma forma padrão para a preparação de planos de arquitetura de projetos de sistemas, incluindo aspectos conceituais tais como processos de negócios e funções do sistema, além de itens concretos como as classes escritas em determinada linguagem de programação, esquemas de bancos de dados e componentes de software reutilizáveis.”



James Rumbaugh



Grady Booch



Ivar Jacobson

No processo de definição da UML, procurou-se aproveitar o melhor das características das notações preexistentes, principalmente das técnicas propostas anteriormente pelos três amigos (essas técnicas eram conhecidas pelos nomes Booch Method, OMT e OOSE). A notação definida para a UML é uma união de diversas notações preexistentes, com alguns elementos removidos e outros elementos adicionados com o objetivo de torná-la mais expressiva.

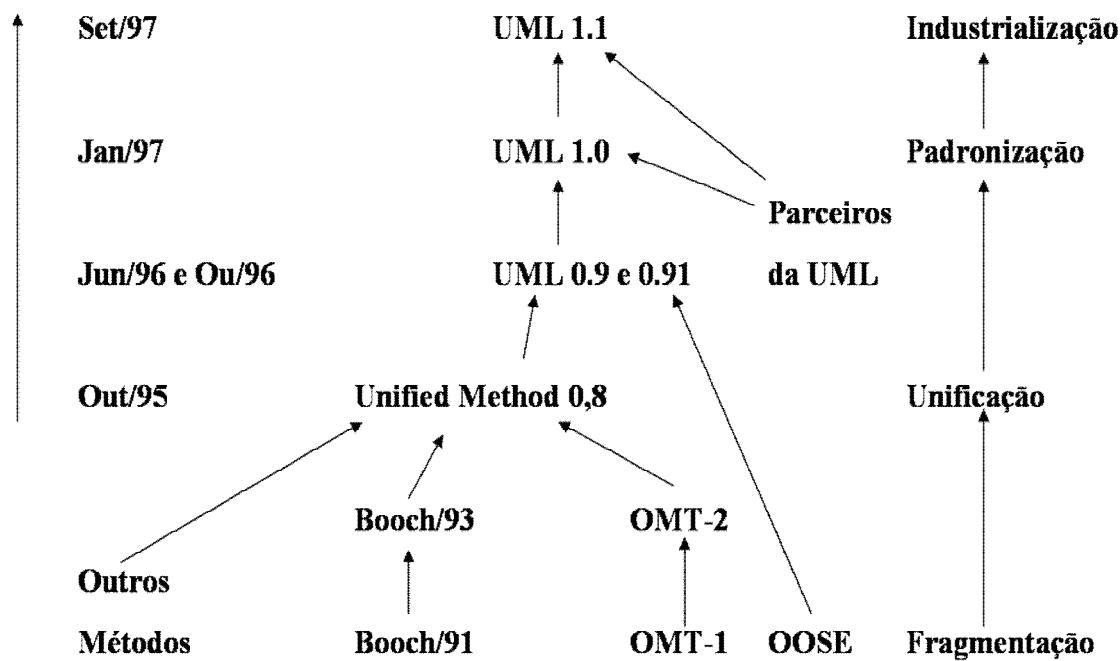
Finalmente, em 1997, a UML versão 1.1 foi aprovada como padrão pelo OMG (ver nossa ATIVIDADE ao final desta Unidade). Desde então, a UML tem tido grande aceitação pela

comunidade de desenvolvedores de sistemas. A sua definição ainda está em desenvolvimento e possui diversos colaboradores. Tanto que, desde o seu surgimento, várias atualizações foram feitas no sentido de torná-la mais clara e útil. Atualmente estamos na versão 2.1.1 da UML, mas ela é tão dinâmica que provavelmente quando você estiver fazendo a Atividade encontre uma versão ainda mais atual. Veja o histórico do UML na figura a seguir.

A UML é uma linguagem visual para modelar sistemas orientados a objetos. Isso quer dizer que a UML é uma linguagem constituída de elementos gráficos utilizados na modelagem que permitem representar os conceitos do paradigma da orientação de objetos. Através dos elementos gráficos definidos nesta linguagem podem-se construir diagramas que representam diversas perspectivas de um sistema.

A UML é uma linguagem de modelagem, não é um método, nem mesmo metodologia !!

Feed Back Públco



Cada elemento gráfico possui uma sintaxe, uma forma predeterminada de desenhar o elemento, e uma semântica que definem o que significa o elemento e para que ele deve ser utilizado. Além disso, conforme veremos mais adiante, tanto a sintaxe quanta a semântica da UML são extensíveis. Essa extensibilidade permite que a UML seja adaptada às características específicas de cada projeto de desenvolvimento.

Pode-se fazer uma analogia da UML com uma caixa de ferramentas. Um construtor usa sua caixa de ferramentas para realizar suas tarefas. Da mesma forma, a UML pode ser vista como uma caixa de ferramentas utilizada pelos desenvolvedores de sistemas para realizar a construção de modelos.

A UML é uma linguagem de modelagem destinada a realizar atividades especiais em artefatos de um sistema complexo de software. Tais atividades são:

Visualizar	Construir
Especificar	Documentar

A UML é independente tanto de linguagem de programação quanto de processos de desenvolvimento. Isso quer dizer que a UML pode ser utilizada para a modelagem de sistemas, não importando qual a linguagem de programação a ser utilizada, ou qual a forma de desenvolvimento adotada. Esse é um fator importante para a utilização da UML, pois diferentes sistemas de software requerem diferentes abordagens de desenvolvimento.

Visões De Um Sistema Pela UML

O desenvolvimento de um sistema de software complexo demanda que seus desenvolvedores tenham a possibilidade de examinar e estudar esse sistema a partir de diversas expectativas. Os autores da UML sugerem que um sistema pode ser descrito por cinco visões interdependentes desse sistema. Cada visão enfatiza aspectos diferentes do sistema (veja a figura a seguir). As visões propostas são as seguintes:

Visão de Casos de Uso

Descreve o sistema de um ponto de vista externo como um conjunto de interações entre o sistema e os agentes externos ao sistema. Esta visão é criada inicialmente e direciona o desenvolvimento das outras visões do sistema.

Visão de Projeto

Enfatiza as características do sistema que dão suporte, tanto estrutural quanto comportamental, às funcionalidades externamente visíveis do sistema.

Visão de Implementação

Abrange o gerenciamento de versões do sistema, construídas através dos agrupamentos de módulos (componentes) e subsistemas.

Visão de Implantação

Corresponde à distribuição física do sistema em seus subsistemas e à conexão entre essas partes.

Visão de Processo

Esta visão enfatiza as características de concorrência (paralelismo), sincronização e desempenho do sistema.

Dependendo das características e da complexidade do sistema, nem todas as visões precisam ser construídas. Por exemplo, se o sistema tiver de ser instalado em um ambiente computacional de processador único, não há a necessidade da visão de implantação.

Outro exemplo: se o sistema for constituído de um único processo, a visão de processo é irrelevante. De forma geral, dependendo do sistema, as visões podem ser ordenadas por grau de relevância.



Estudo Complementar

Veja o rico material sobre o RUP e especificamente sobre visões em:

http://www.wthreeex.com/rup/process/workflow/requirem/co_ucv.htm





Atividades

Visite o site da OMG – Object Management Group (<http://www.uml.org/>) é um consórcio internacional formado por empresas tais como: HP, IBM, Oracle, Microsoft e outras, e possui muitas informações interessantes.

Uma delas é a Especificação da Linguagem de Modelagem Unificada totalmente gratuito da definição completa da UML. Tente baixar esse arquivo. Esse documento possui um sumário, semântica, guia da notação e extensões da linguagem UML.



UNIDADE 18

Objetivo: Descrever com mais detalhes o Paradigma da Orientação de Objetos.

O Paradigma da Orientação de Objetos

Uma definição simples de paradigma seria “uma forma de abordar um problema”. Pode-se dizer, então, que o termo Paradigma da Orientação a Objetos é uma forma de abordar um problema. Há alguns anos, Alan Kay, um dos pais do Paradigma da Orientação a Objetos, formulou a chamada “analogia biológica”.

Nessa analogia, ele imaginou como seria um sistema de software que funcionasse com um ser vivo. Nesse sistema, cada célula interagiria com outras células através do envio de mensagens para realizar um objetivo comum. Adicionalmente, cada célula se comportaria como uma unidade autônoma.

De uma forma mais geral, Kay pensou em como construir um sistema de software a partir de agentes autônomos que interagem entre si. Ele, então, estabeleceu os seguintes princípios da orientação a objetos:

- Qualquer coisa é um objeto.
- Objetos realizam tarefas através da requisição de serviços a outros objetos.
- Cada objeto pertence a uma determinada classe. Uma classe agrupa objetos similares.
- A classe é um repositório para o comportamento associado ao objeto.
- Classes são organizadas em hierarquias.

Paradigma Orientação De Objetos versus Paradigma Estruturado

Como já vimos anteriormente, antes da orientação de objetos, outro paradigma era utilizado na modelagem de sistemas: o Paradigma Estruturado. Nesse paradigma, os elementos principais são dados e processos. Processos agem sobre dados para que um objetivo seja alcançado.

Por outro lado, no Paradigma da Orientação de Objetos, há um elemento, o objeto - uma unidade autônoma - que contém seus próprios dados que são manipulados pelos processos definidos para o objeto e que interage com outros objetos para alcançar também um objetivo.

É o Paradigma da Orientação de Objetos que os seres humanos tipicamente utilizam no cotidiano para a resolução de problemas. Uma pessoa atende a mensagens (requisições) para realizar um serviço. Por sua vez essa mesma pessoa envia mensagens a outras para que estas realizem outros serviços. Por que não aplicar essa mesma forma de pensar a modelagem de sistemas?

Vejamos o exemplo geral a seguir para revisar os conceitos apresentados anteriormente:

João quer comprar uma pizza, e por estar ocupado, pede a pizza por telefone. João informa ao atendente Luís seu nome, a pizza e o seu endereço. Por sua vez Luís avisa ao pizzaiolo Antonio qual a pizza a ser feita. Uma vez feita a pizza Luís chama Roberto, o motoboy, para entregá-la na casa do João.

O objetivo de João foi atingido através da colaboração de diversos agentes, que são denominados objetos. Estes objetos foram: Luís, Antonio e Roberto. Embora todos tenham trabalhado em suas funções de forma individual, alcançaram o objetivo cooperando conjuntamente. O comportamento do pizzaiolo Antonio, embora seja um do melhores da sua classe, é o mesmo esperado de qualquer um de sua profissão (fazer pizza). Logo Antonio é um objeto da classe Pizzaiolo. E todos os objetos pertencem a classes maiores por serem seres humanos, animais, mamíferos e assim por diante (hierarquias).

Classes E Objetos

Na terminologia de orientação de objetos coisas do mundo real como um cliente, uma loja, uma venda, um pedido de compra, um fornecedor, esta apostila, são denominados **objetos**.

Os objetos possuem características ou propriedades que são seus **atributos**. Esses atributos identificam o estado de um objeto. Um atributo é uma abstração do tipo de dados ou estado que os objetos da classe possuem.

Nos, seres humanos, costumamos agrupar objetos. Provavelmente, fazemos isso para tentar gerenciar a complexidade de entender as coisas do mundo real. Realmente, é bem mais fácil entender a ideia “cavalo” do que entender todos os cavalos que existem no mundo.

Na terminologia da orientação de objetos, cada ideia é denominada classe de objetos, ou simplesmente classe. Uma classe é uma descrição dos atributos e serviços comuns a grupo de objetos. Sendo assim, pode-se entender uma classe como sendo um molde a partir do qual objetos são construídos. Ainda sobre terminologia, diz-se que um objeto é uma instância de uma classe.

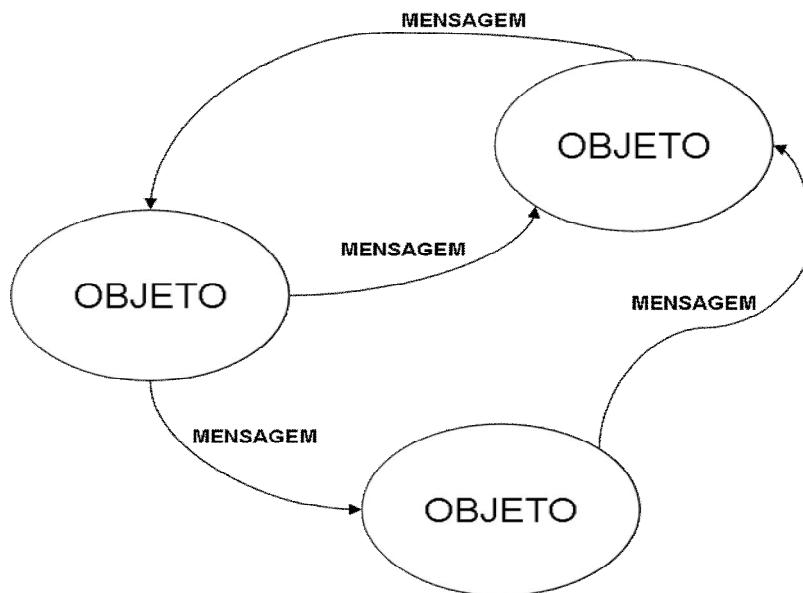
É importante notar que uma classe é uma abstração das características de um grupo de coisas do mundo real. Na maioria das vezes, as coisas do mundo real são muito mais complexas para que todas as suas características sejam representadas em uma classe. Além disso, para fins de modelagem de um sistema, somente um subconjunto de características pode ser relevante. Portanto, uma classe representa uma abstração das características relevantes do mundo real.

Mensagens

Para que uma operação em um objeto seja executada, deve haver um estímulo enviado a esse objeto. Se um objeto for visto como uma entidade ativa que representa uma abstração de algo do mundo real, então faz sentido dizer que tal objeto pode responder a estímulos a ele enviados, assim como faz sentido dizer que seres vivos reagem a estímulos que eles recebem.

Independentemente da origem do estímulo, quando ele ocorre, diz-se que o objeto em questão está recebendo uma **mensagem** requisitando que ele realize alguma operação.

Quando se diz que objetos de um sistema estão trocando mensagens significa que esses objetos estão enviando mensagens uns aos outros com o objetivo de realizar alguma tarefa dentro do sistema no qual eles estão inseridos.



Estudo Complementar

http://pt.wikipedia.org/wiki/Orienta%C3%A7%C3%A3o_a_objeto





Atividades

Como tipicamente identificamos e diferenciamos objetos por seus atributos, tente identificar o objeto a seguir dado os seguintes atributos:

Nome

Endereço

Data de nascimento

Especialização

CRM



UNIDADE 19

Objetivo: Continuar a definir conceitos básicos do paradigma da Orientação de Objetos.

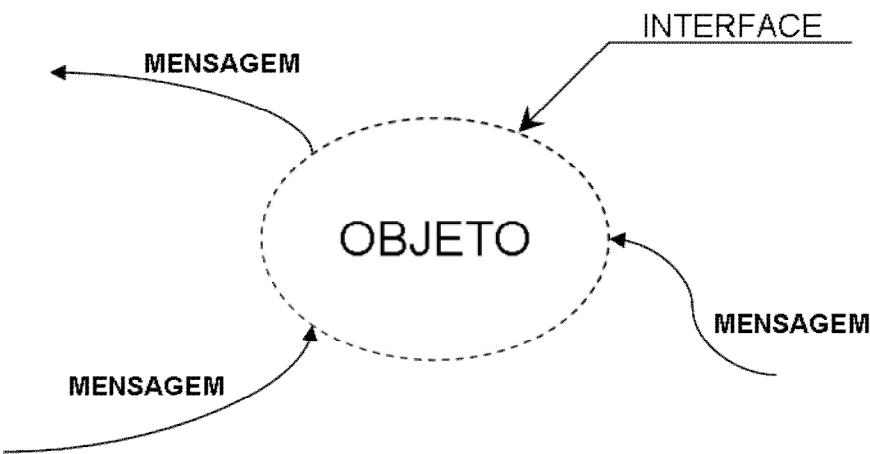
Encapsulamento, Polimorfismo e Herança

Objetos possuem comportamento. O termo comportamento diz respeito a **operações** realizadas por um objeto e também ao modo pelo qual essas operações são executadas. O mecanismo de **encapsulamento** é uma forma de restringir o acesso ao comportamento interno de um objeto. Um objeto que precise da colaboração de outro objeto para realizar alguma tarefa simplesmente envia uma mensagem a este último. O método que o objeto requisitado usa para realizar a tarefa não é conhecido dos objetos requisitantes.

Dentro desse conceito visualizamos o objeto como uma “caixa preta”. Quem descreve o comportamento de um objeto é a sua classe. Um objeto possui uma **interface**. A interface de um objeto é o que ele conhece e o que ele sabe fazer, sem descrever como o objeto conhece o que faz. A interface de um objeto define os serviços que ele pode realizar e consequentemente as mensagens que ele recebe.

Uma interface pode ter várias formas de implementação. Mas, pelo princípio do encapsulamento, a implementação de um objeto requisitado não importa para o objeto requisitante. Através do encapsulamento, a única coisa que um objeto precisa saber para pedir a colaboração de outro objeto é conhecer sua interface. Nada mais. Isso contribui para a autonomia dos objetos.

Cada objeto envia mensagens a outros objetos para realizar certas tarefas, sem se preocupar em como as tarefas são realizadas. A aplicação da abstração, nesse caso, está em esconder os detalhes de funcionamento interno de um objeto.



Polimorfismo

O polimorfismo indica a capacidade de abstrair várias implementações diferentes em uma única interface. Um bom exemplo para explicar este conceito seria o controle remoto. Embora ele seja normalmente fabricado especificamente para um tipo de aparelho, existem controles remotos considerados universais. Portanto, mesmo com um controle remoto de outro fabricante é possível acionar o aparelho de outro fornecedor.

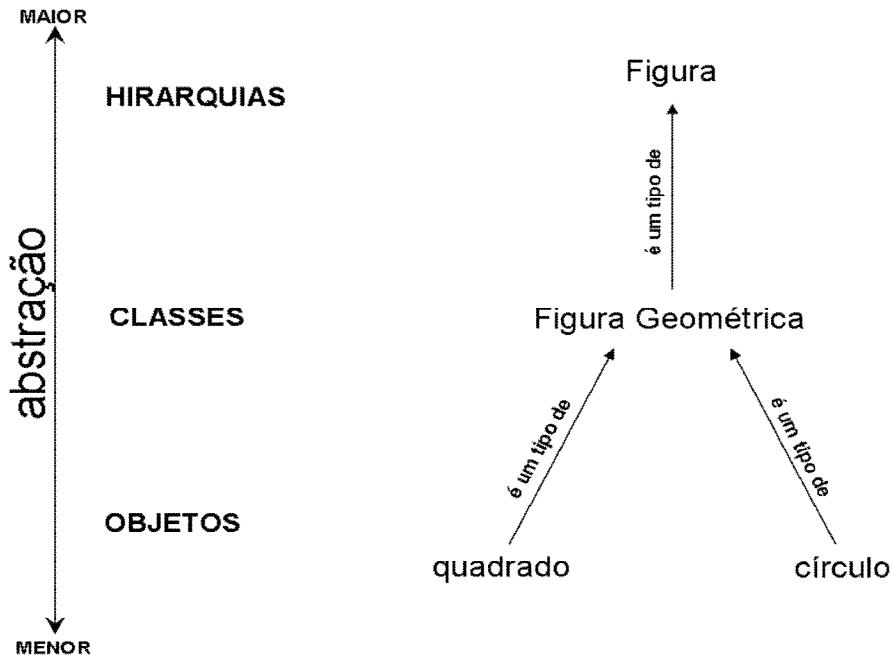
Esse é um exemplo de aplicação do princípio do polimorfismo. Note mais uma vez que a abstração também é aplicada aqui: um objeto pode enviar a mesma mensagem para objetos semelhantes, mas que implementam a sua interface de formas diferentes.

Herança

A herança é outra forma de abstração utilizada na orientação de objetos. As características e o comportamento comuns a um conjunto de objetos podem ser abstraídos em uma classe. A herança pode ser vista como um nível de abstração acima da encontrada entre classes e objetos.

Na herança, classes semelhantes são agrupadas em hierarquias. Cada nível de uma hierarquia pode ser visto como um nível de abstração. Cada classe em um nível da hierarquia herda as características das classes nos níveis acima. Esse mecanismo facilita o compartilhamento comum entre um conjunto de classes semelhantes. Além disso, as

diferenças ou variações de uma classe em particular podem ser organizadas de forma mais clara.



Estudo Complementar

<http://www.ic.unicamp.br/~cmrubira/aacesta/java/javatut10.html>





Atividades

Identifique paralelos entre as seguintes características de uma célula e os conceitos da orientação a objetos descritos nesta unidade:

Mensagens são enviadas de uma célula a outra através de receptores químicos.

Células têm uma fronteira, a membrana celular. Cada célula tem um comportamento interno que não é visível de fora.

As células podem se reagrupar para resolver problemas ou para realizar uma função.



Atividades

Antes de dar continuidades aos seus estudos é fundamental que você acesse sua SALA DE AULA e faça a Atividade 2 no “link” ATIVIDADES.



UNIDADE 20

Objetivo: Apresentar as principais linguagens Orientada a Objetos no mercado e o seu histórico.

Linguagens Orientada a Objetos

A primeira linguagem com os conceitos de orientação a objetos foi a linguagem **Simula**. Foi iniciado o desenvolvimento dessa linguagem em 1962 na Noruega, por Olé-Johan Dahl e Kristen Nygaard, e curiosamente muito antes dos métodos estruturais.

Baseada na linguagem **Algol 60** seu desenvolvimento teve três fases: **Simula 0** (1962-1963), **Simula I** (1963-1965) e **Simula 67** (1966-1967). Com esse projeto surgiram os primeiros conceitos de classes, com seus atributos e métodos, encapsulamento e herança.

Simula 67

Influenciou a primeira linguagem totalmente orientada a objetos que viria seguir, a **SmallTalk**. Essa linguagem foi desenvolvida por Alan Kay, no início da década de 70, no famoso e gerador das principais tecnologias usadas hoje, o Centro de Pesquisas da Xerox, em Palo Alto.

Disponibilizada ao público no início dos anos 80, **SmallTalk** solidificou para a comunidade os conceitos de classe, objeto, atributo, método, encapsulamento, herarquia, herança e mensagem. Por ser uma linguagem puramente orientada a objetos, tudo em seu ambiente é visto como um objeto, e qualquer comunicação entre objetos é feita por mensagens.

O **SmallTalk** continua sendo usado comercialmente por vários fornecedores, com nomes complementares distintos. Durante estas últimas décadas, a orientação a objetos amadureceu. Com pequeno esforço implementam-se os modelos Orientado a Objeto nos bancos de dados relacionais já existentes. Assim, não há a necessidade de se mexer nas bases de dados mais antigas.

Podemos classificar as linguagens orientadas a objeto como **híbridas** e **puras**. Uma linguagem híbrida é criada a partir da ampliação de uma linguagem procedural, de origem na análise estruturada, permitindo a implementação da orientação a objetos. Ou seja, a linguagem mantém a programação estruturada e acrescentada a orientada a objetos.

A linguagem pura só permite implementação baseada na programação orientada a objetos. As únicas linguagens puras, atualmente no mercado, são: **SmallTalk** e **Eiffel**.

O **Eiffel** não possui linguagem de origem. Foi desenvolvido por Bertrand Meyer, em 1986, na Interactive Software Engineering (ISE), seguindo os estritos princípios de orientação a objetos. É muito apreciada por acadêmicos exatamente por permitir o aprendizado puro dos conceitos de OO (Orientado a Objetos).

A linguagem **C++** é uma evolução da linguagem **C** acrescentando suporte ao paradigma de programação orientada a objetos. Portanto, é uma linguagem híbrida. Tomou por base o estilo de programação usado na linguagem **Simula**.

Tornou-se a linguagem nativa de alguns ambientes integrados de desenvolvimento como: **C++ Builder** (Borland Corporation) e **Visual C++** (Microsoft) entre outros.

Object Pascal

Consiste numa evolução da linguagem **Pascal**. A Borland foi responsável direta por essa evolução. No início dos anos 80, ela lançou o **Turbo Pascal**. Mais tarde, com o surgimento do Windows, surgiu o Turbo Pascal for Windows e um pouco depois, o **Borland Pascal**, considerada a primeira versão do Object Pascal. Hoje, é utilizada como linguagem nativa Borland Delphi.

Java

Foi desenvolvido como um subconjunto da linguagem **C++**, pela Sun Microsystems, e lançada em 1995. Em virtude de seu uso em larga escala, em aplicações para Internet, Intranets e Extranets, tornou-se rapidamente popular. Seus códigos-fontes são compilados e transformados em arquivos-objetos, chamados bytecodes.

Esses bytecodes são executados por interpretadores **Java**, desenvolvidos para cada plataforma de hardware/software, funcionando como Máquinas Virtuais – Java Virtual Machine. Dessa forma, a linguagem consegue trabalhar com uma programação multiplataforma.

Existem dois tipos de bytecodes: aplicações **Java** e **Applets**. No primeiro caso, a aplicação possui acesso completo à máquina, manipulando memória e acessando arquivos. No caso dos **Applets**, estes são tipicamente interpretados por browsers, devidamente capacitados, e possuem restrições de acesso à memória e arquivos.

Tornou-se linguagem nativa de alguns ambientes integrados de desenvolvimento, como: **JBuilder** (Borland Corporation), **JDeveloper** (Oracle Corporations), **VisualAge for Java** (IBM), entre outros.



Estudo Complementar

http://pt.wikipedia.org/wiki/Linguagem_de_programa%C3%A7%C3%A3o



Atividades

Através do site <http://www.levenez.com/lang/history.html>, veja se você consegue identificar a maioria das linguagens citadas nesta Unidade.



UNIDADE 21

Objetivo: Inicializar os conceitos básicos que envolvem o Diagrama de Casos de Uso.

Diagrama de Casos de Uso

A maior dificuldade em modelarmos um sistema não está nos diagramas que temos que desenhar, no código que devemos criar ou nas bases de dados que devemos projetar. Na realidade está nos requisitos que devemos gerenciar.

O modelo de casos de uso é uma representação das funcionalidades externamente observáveis do sistema e dos elementos externos ao sistema que interagem com ele. Este modelo é parte integrante da especificação de requisitos.

Na verdade, o Modelo de Casos de Uso molda os requisitos funcionais do sistema. O diagrama da UML utilizado na modelagem de casos de uso é o **diagrama de casos de uso**.

A técnica de modelagem através de casos de uso foi idealizada por um famoso engenheiro de software sueco, Ivar Jacobson (um dos três amigos), na década de 70, enquanto trabalhava no desenvolvimento de um sistema na empresa de telefonia Ericson. Mais tarde, Jacobson incorporou essa técnica a um processo de desenvolvimento de software denominado Objetory. Posteriormente, ele se uniu a Booch e a Rumbaugh, e a notação de casos de uso foi incorporada à Linguagem de Modelagem Unificada.

Desde então, esse modelo vem se tornando cada vez mais popular para realizar a documentação de requisitos funcionais de uma aplicação. Devido à sua notação gráfica simples e descrição em linguagem natural, o que facilita a comunicação de desenvolvedores e usuários.

Este modelo é um dos mais importantes da UML. Ele direciona diversas tarefas posteriores ao ciclo de vida do sistema de software. Além disso, o modelo de casos de uso força os

desenvolvedores a moldarem o sistema de acordo com o usuário, e não o usuário de acordo com o sistema.

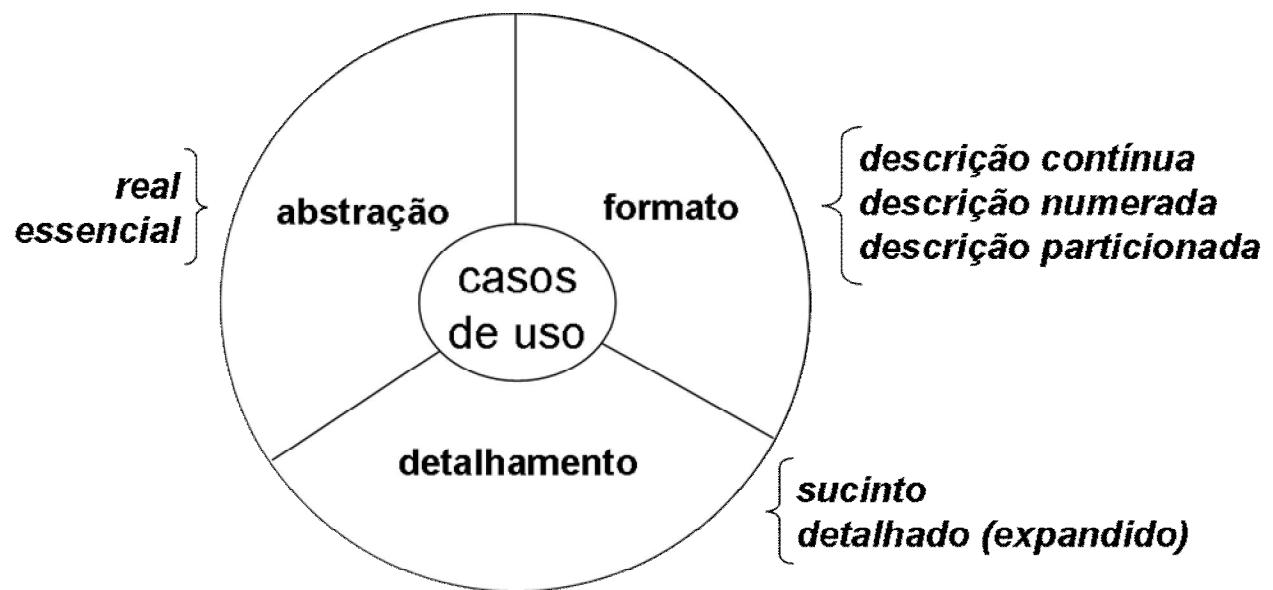
O modelo de casos de uso é composto de: casos de uso, atores e relacionamento entre estes. Vejamos maiores detalhes de cada um a seguir.

Casos De Uso

Um caso de uso é a especificação de uma sequência de interações entre um sistema e os agentes externos que utilizam esse sistema. Um caso de uso deve definir o uso de uma parte da funcionalidade de um sistema, sem revelar a estrutura e os comportamentos internos desse sistema. Um modelo de casos de uso típico contém vários casos de uso.

Um caso de uso representa quem faz o que com o sistema, sem considerar o comportamento interno do sistema.

Cada caso de uso deve ser definido através da descrição narrativa das interações que ocorrem entre os elementos externos e o sistema. A UML não define o formato e o grau de abstração a serem utilizados na descrição de um caso de uso.



Veremos na próxima unidade o detalhamento de cada um dos itens apresentados na figura acima.



Estudo Complementar

http://www.wthreeex.com/rup/process/artifact/ar_uc.htm

http://www.wthreeex.com/rup/process/artifact/ar_ucmgl.htm



Atividades

Através do roteiro do site <http://www.wthreeex.com/rup/manuals/ucmgl/index.htm>
tente desenvolver o seu próprio Diagrama de Casos de Uso.

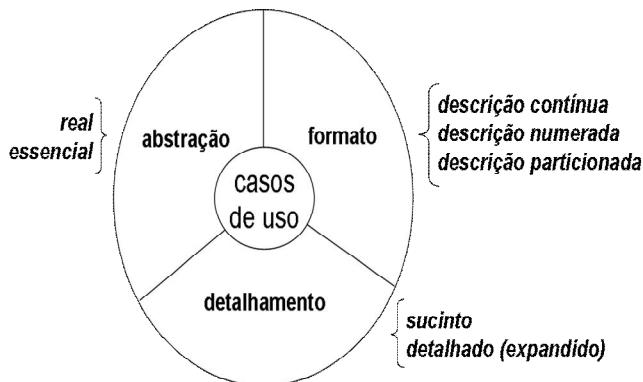


UNIDADE 22

Objetivo: Detalhar os diversos itens que um Caso de Uso possui e suas definições.

Casos de Uso: Formato, Detalhamento e Abstração

Casos De Uso – Formato



Quanto ao FORMATO comumente são utilizadas a descrição contínua, a descrição numerada e a descrição particionada. Esses tipos de narrativa são apresentados a seguir, com o exemplo correspondente ao caso de uso de saque de determinada quantia em um caixa eletrônico de um sistema bancário.

No formato de descrição contínua a narrativa é feita através de um texto livre. Como exemplo considere o caso de uso **Realizar Saque** em um caixa eletrônico:

O Cliente chega ao caixa eletrônico e insere seu cartão. O sistema requisista a senha do Cliente. Após o Cliente fornecer sua senha e esta ser validada, o Sistema exibe as opções de operações possíveis. O Cliente opta por realizar um saque. Então o Sistema requisita o total a ser sacado. O Sistema fornece a quantia desejada e imprime o recibo para o Cliente.

No formato de descrição numerada, a narrativa é descrita através de uma série de passos numerados. Considere como exemplo o mesmo caso de uso **Realizar Saque**:

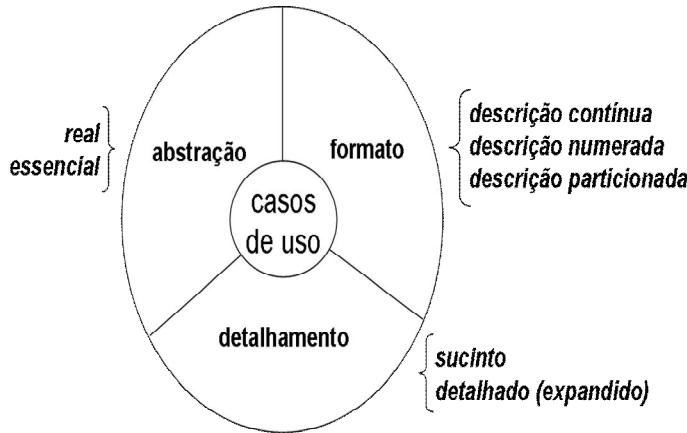
1. *Cliente insere seu cartão no caixa eletrônico.*
2. *Sistema apresenta solicitação de senha.*
3. *Cliente digita senha.*

4. Sistema exibe menu de operações disponíveis.
5. Cliente indica que deseja realizar saque.
6. Sistema requisita quantia a ser sacada.
7. Cliente retira a quantia e o recibo.

O estilo de descrição particionada tenta prover alguma estrutura à descrição de casos de uso. A sequência de interações entre o ator e o sistema é particionada em duas colunas, uma para o ator e outra para o sistema:

CLIENTE	SISTEMA
Insere seu cartão no caixa eletrônico.	
	Apresenta solicitação de senha.
Digita senha.	
	Exibe menu de operações disponíveis.
Solicita realização de saque.	
	Requisita quantia a ser sacada.
Retira a quantia e o recibo.	

CASOS de USO – DETALHAMENTO



O grau de detalhamento a ser utilizado na descrição de um caso de uso pode variar desde o mais sucinto até a descrição envolvendo vários detalhes (expandido). Um caso de uso sucinto descreve as interações sem muitos detalhes. Um caso de uso expandido descreve as interações em detalhes.

CASOS de USO – ABSTRAÇÃO

O grau de abstração de um caso de uso diz respeito à existência ou não de menção à tecnologia a ser utilizada na descrição desse caso de uso. Em relação ao grau de abstração, um caso de uso pode ser real ou essencial.

Um caso de uso essencial é abstrato e não faz menção à tecnologia a ser utilizada. Note que o exemplo a seguir de caso de uso essencial e numerado é completamente desprovido de características tecnológicas. Esse caso de uso vale para a situação em que o sistema tem mais de uma interface para a mesma funcionalidade. Por exemplo, uma interface no site na Internet e outra interface via telefone celular.

1. *Cliente fornece sua identificação.*
2. *Sistema identifica o usuário.*
3. *Sistema fornece operações disponíveis.*
4. *Cliente solicita o saque de uma determinada quantia.*
5. *Sistema fornece a quantia desejada da conta do Cliente.*
6. *Cliente recebe dinheiro e recibo.*

Diferentemente, em um caso de uso real, as descrições das interações citam detalhes da tecnologia a ser utilizada na implementação. A descrição em um grau de abstração real se compromete com a solução de projeto (tecnologia) a ser utilizada para implementar o caso de uso.

É uma boa ideia utilizar a “**regra prática dos 100 anos**” para identificar se um caso de uso contém algum detalhe de tecnologia. Questione se a narrativa seria válida tanto 100 anos atrás, como daqui a 100 anos. Se a resposta for positiva, então muito provavelmente esse caso de uso é essencial. Caso contrário, trata-se de um caso de uso real.



Estudo Complementar

Veja um bom resumo dos principais tópicos relacionados até aqui no site:

www.dimap.ufrn.br/~flavia.delicato/ModeloRequisitosCasosdeUSos.pdf



Atividades

Com base no documento acima realize um resumo sintético dos principais tópicos abordados.



UNIDADE 23

Objetivo: Detalhar os diversos elementos que um Caso de Uso possui e suas definições.

Caso de Uso – Atores e Relacionamentos

Na terminologia da UML, qualquer elemento externo que interage com o sistema é denominado ator. Vamos detalhar melhor essa definição. O termo “externo” indica que atores não fazem parte do sistema. E o termo “interage” significa que um ator troca (envia e/ou recebe) informações com o sistema.

CATEGORIA DE ATORES	EXEMPLOS
Pessoas	Empregado, cliente, gerente, almoxarife, vendedor
Organizações	Empresa fornecedora, Agência de Turismo, Administradora de Cartões
Outros Sistemas	Sistema de Cobrança, Sistema de Estoque, Sistema de Vendas
Equipamentos	Leitora de Código de Barras, Sensor, Plotter, catraca eletrônica

Levando em conta que um ator é um papel representado no sistema, o nome dado a esse ator, portanto, deve lembrar o seu papel, em vez de lembrar quem o representa. Exemplos de bons nomes para atores: Cliente, Estudante, Fornecedor, etc. Exemplos de nomes inadequados para atores: João, Fornecedor, ACME, etc.

CASO de USO – RELACIONAMENTOS

A UML define vários tipos de relacionamentos no modelo de casos de uso: comunicação, inclusão, extensão e generalização. Vamos dividir em blocos para ficar mais fácil didaticamente:

INTERAÇÕES	COMUNICAÇÃO	INCLUSÃO	EXTENSÃO	GENERALIZAÇÃO
Caso de Uso e Caso de Uso		*	*	*
Autor e Autor				*
Autor e Casos de Uso	*			

Comunicação

Representa a interação do ator com o caso de uso, ou seja, a comunicação entre atores e casos de uso, por meio de envio e recebimento de mensagens.

A comunicação entre casos de uso são sempre binárias, ou seja, envolvem apenas dois elementos. Representam o único relacionamento possível entre atores e casos de uso. Por exemplo: O ator Correntista envia e recebe mensagens do Caso de Uso Calcular Empréstimo Pessoal, por um relacionamento de comunicação.

Generalização

Ocorre entre Casos de Uso ou entre Atores. É considerado quando temos dois elementos semelhantes, mas com um deles realizando algo a mais. Costuma ser comparado ao relacionamento de extensão, com uma variação do comportamento normal sendo descrita sem muito rigor. Segue o mesmo conceito da orientação a objetos.

Por exemplo: num Caso de Uso no qual C2 herda de C1, significa dizer que temos C1 como um Caso de Uso mais genérico e C2 mais específico. Outro exemplo: podemos criar um ator genérico **Aluno** e especializá-lo nos atores **Aluno Matriculado** e **Aluno Ouvinte**.

Extensão

Um relacionamento extensão entre Casos de Uso indica que um deles terá seu procedimento acrescido, em um ponto de extensão, de outro Caso de Uso, identificado como base.

Os pontos de extensão são rótulos que aparecem nos cenários de Caso de Uso base. É permitido colocar diversos pontos de extensão num mesmo Caso de Uso, inclusive repetir um mesmo ponto de extensão.

Um Caso de Uso de extensão é muito utilizado para:

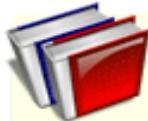
1. Expressar rotinas de exceção ou para expressar o desmembramento de um Caso de Uso.
2. Separar um comportamento obrigatório de outro opcional.
3. Separar um trecho do Caso de Uso que será executado apenas em determinas condições.
4. Separar trechos que dependam da interação com um determinado ator.

Inclusão

Quando existem cenários cujas ações servem a mais de um Caso de Uso deve-se utilizar um relacionamento de inclusão. Indica que um deles terá seu procedimento copiado num local especificado no outro Caso de Uso, identificado como base.

Textualmente, um relacionamento de inclusão, dentro da descrição de um Caso de Uso base, é representado com o termo **Include** seguido de seu nome. Mais do que evitar a cópia de trechos idênticos ganhamos tempo de modelagem e também de implementação, e consequentemente de manutenção, ao trabalhar com Casos de Uso de inclusão.

Exemplo: Validar Matrícula é útil para Casos de Uso como Renovar Matrícula de Aluno, Emitir Histórico Escolar, Lançar Notas de Provas, entre outros.



Estudo Complementar

<http://celodemelo.wordpress.com/2007/03/17/entendendo-o-diagrama-de-casos-de-uso/>

http://www.macoratti.net/net_uml2.htm



Atividades

Como você explicaria a diferença existente entre os diversos tipos de relacionamentos de casos de uso?

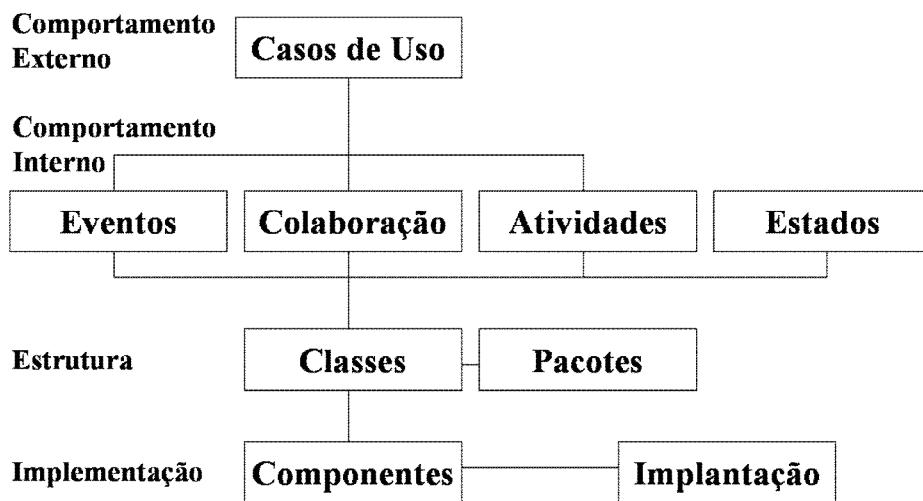


UNIDADE 24

Objetivo: Apresentar os Diagramas da UML, suas categorias e diferenças com versões anteriores.

Visão Geral dos Diagramas da UML e as suas categorias

Diretamente derivada dos conceitos de programação e do projeto orientado por objeto, a análise orientada a objetos é certamente a mais destacada das abordagens de análise de sistemas.



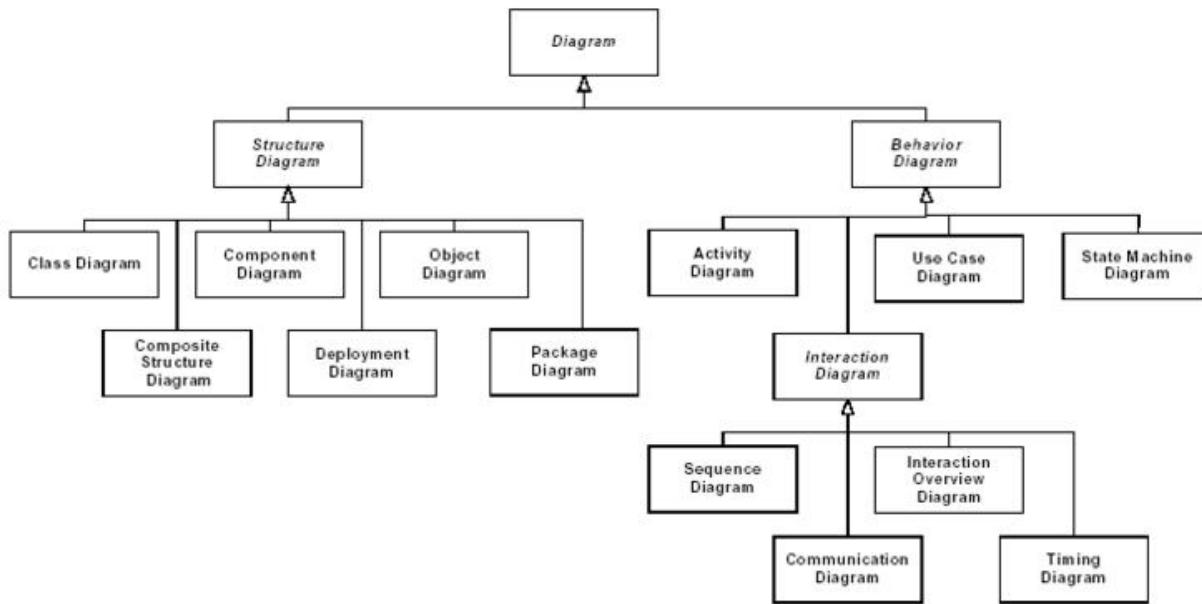
Baseada na decomposição do sistema de acordo com os objetos (entidades de dados), essa abordagem oferece como principais benefícios os seguintes fatores:

Mantém a modelagem do sistema e, consequentemente, a automação do mesmo o mais próximo possível de uma visão conceitual do mundo real.

Baseia a decomposição e modelagem do sistema nos **dados**, que é o elemento mais estável de todos aqueles que compõem um Sistema de Informação.

Existem atualmente na UML 13 tipos de diagrama, divididos em duas categorias: **Diagramas Estruturais** e **Comportamentais**.

A função dos Diagramas Estruturais é a de mostrar as características do sistema que não mudam ao longo do tempo. Já os Diagramas Comportamentais mostram como o sistema responde às requisições ou como o mesmo evolui durante o tempo. Veja a figura a seguir:



Diagramas Estruturais (Ou Estáticos)

- Diagrama de classes
- Diagrama de estrutura composta
- Diagrama de componentes
- Diagrama de implantação
- Diagrama de objetos
- Diagrama de pacotes

Diagramas Comportamentais (Ou Dinâmicos)

- Diagrama de atividades
- Diagrama de Caso de Uso
- Diagrama de máquina de estados

Diagramas De Interação Composto Por:

- Diagrama de sequência
- Diagrama de comunicação ou colaboração
- Diagrama de visão geral
- Diagrama de temporal

Mostraremos na tabela a seguir uma relação dos diagramas da versão atual do UML comparado com a versão 1.4:

DIAGRAMAS DA UML 1.4	UML ATUAL
Atividades	Atividades
Caso de Uso	Caso de Uso
Classes	Classes
Colaboração	Comunicação
Componentes	Componentes
Gráfico de Estado	Máquina de Estados
Implantação	Implantação

Objetos	Objetos
Sequência	Sequência
-	Pacotes
-	Estrutura Composta
-	Visão Geral
-	Temporal

Como o objetivo principal deste curso não será a de Modelagem de Sistemas UML, veremos somente os diagramas mais significativos e usados mais frequentemente na Análise de Sistemas.



Estudo Complementar

<http://www.visual-paradigm.com/VPGallery/diagrams/index.html>

<http://www.agilemodeling.com/artifacts/>

http://en.wikipedia.org/wiki/Unified_Modeling_Language



Atividades

Realize uma pesquisa na Internet e tente verificar quais os diagramas da UML são os mais utilizados no mercado.



UNIDADE 25

Objetivo: Visualizar exemplos de dois dos diagramas utilizados no mercado.

Diagrama de Classes e Diagrama de Estrutura

Diagrama de Classes apresenta elementos conectados por relacionamentos. Usado para exibir entidades do mundo real, além de elementos de análise e projeto. Praticamente foi uma derivação do DER da Análise Estruturada que vimos na Unidade 12. Podemos ver um exemplo abaixo:

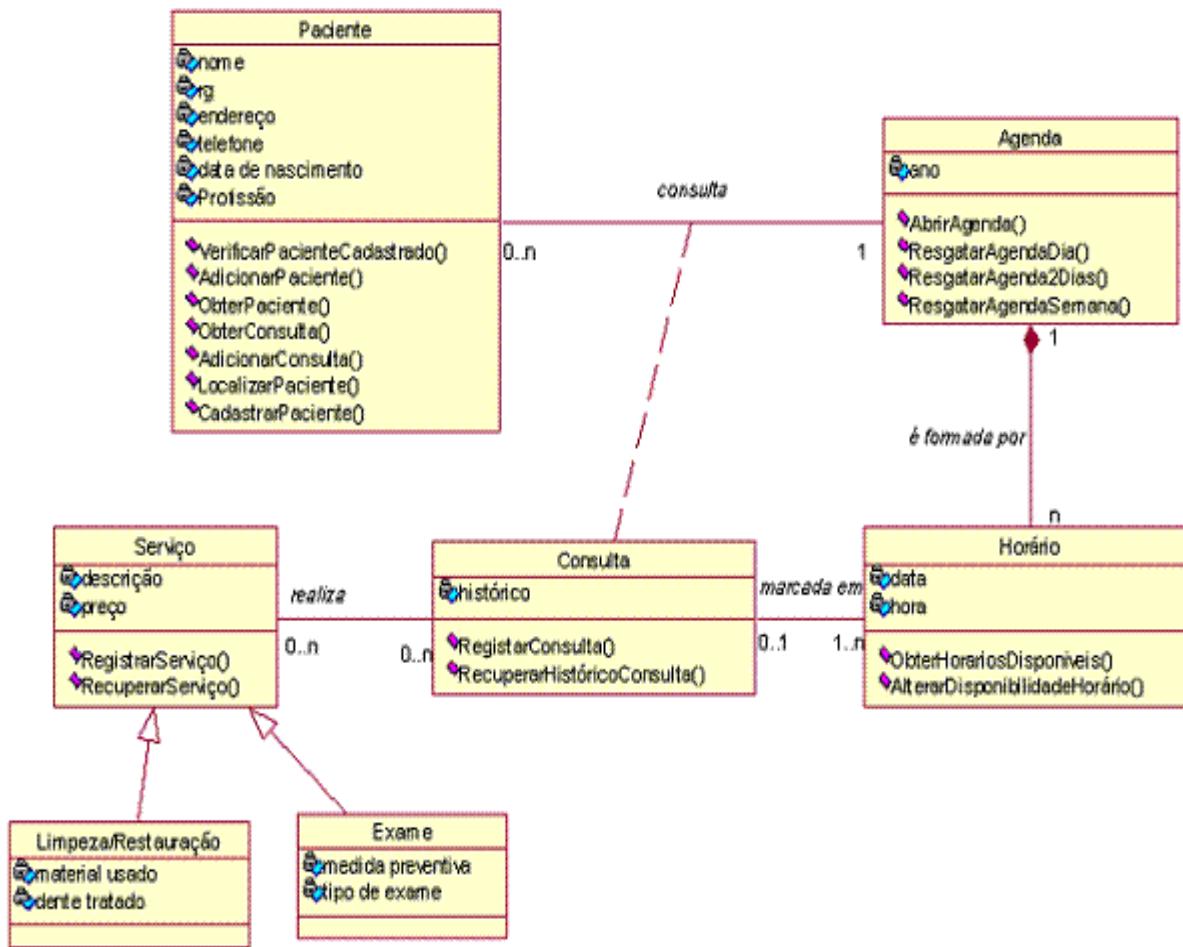
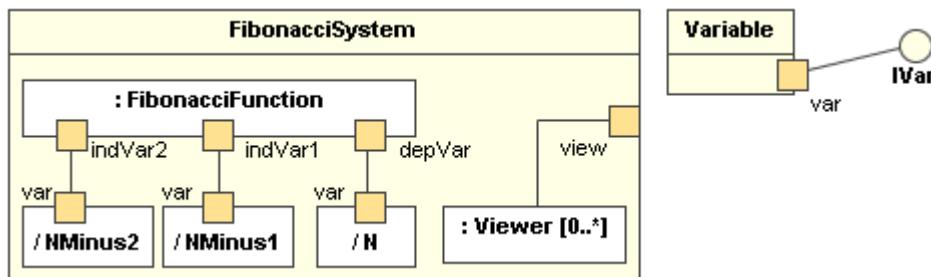


Diagrama de Estrutura, também chamado de Estrutura Composta, é usado para mostrar a composição de uma estrutura. Útil em estruturas compostas de estruturas complexas ou em projetos baseados em componentes.



Estudo Complementar

http://www.wthreeex.com/rup/process/modguide/md_bclsd.htm

http://pt.wikipedia.org/wiki/Diagrama_de_classes

http://pt.wikipedia.org/wiki/Diagrama_de_estrutura_composta



Atividades

Com base nos sites mencionados anteriormente onde você aplicaria mais adequadamente os diagramas desta unidade?



UNIDADE 26

Objetivo: Visualizar exemplos de dois dos diagramas utilizados no mercado.

Diagrama de Componentes e Diagrama de Instalação

Diagrama de Componentes mostra as dependências entre componentes de software, apresentando suas interfaces.

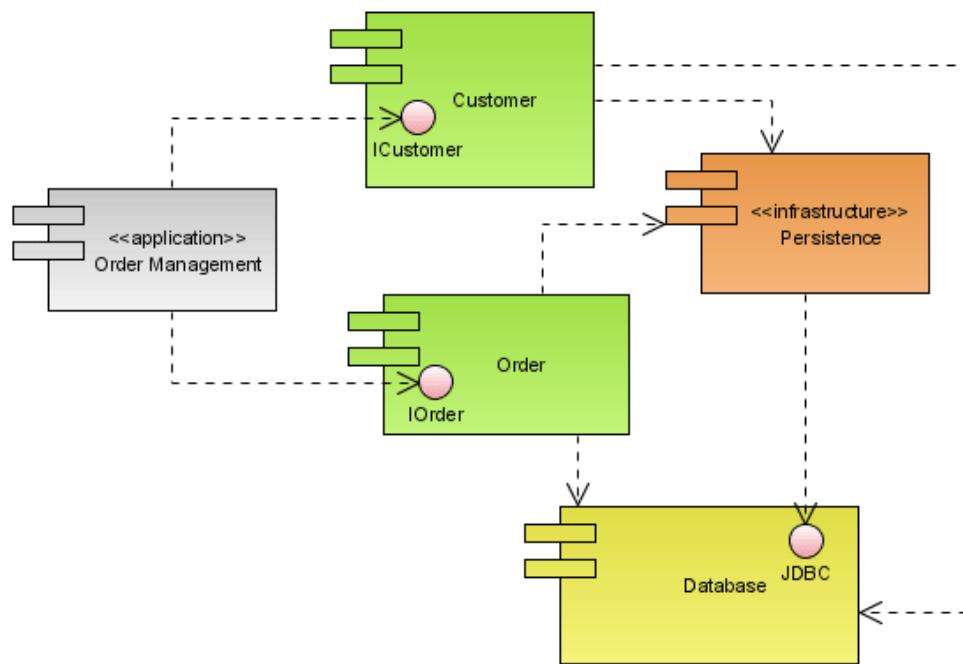
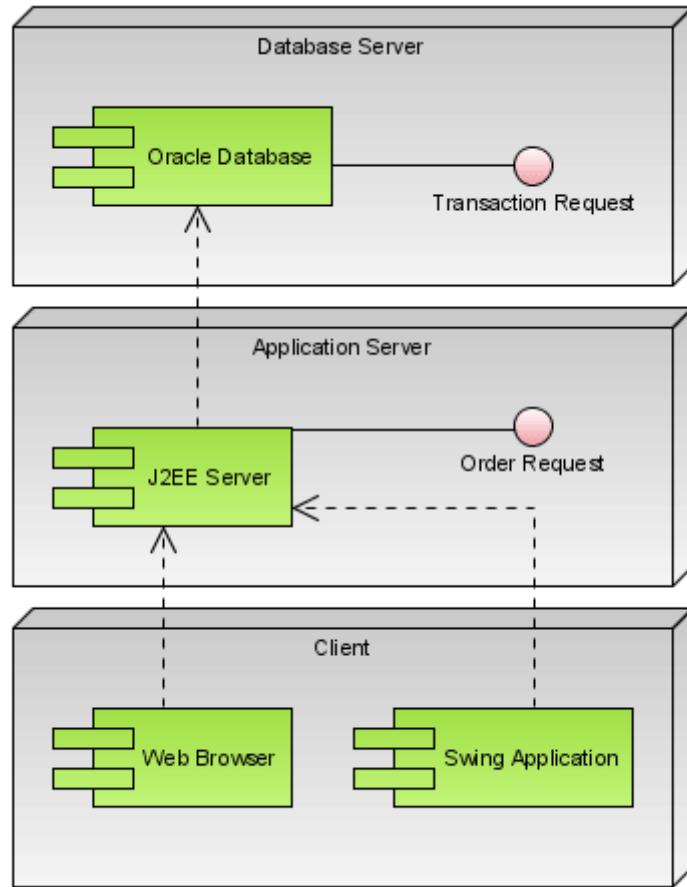


Diagrama de Instalação, também chamado de Diagrama de Implantação, mostra a arquitetura do sistema em tempo de execução, as plataformas de hardware, artefatos de software e ambientes de software como Sistemas Operacionais e máquinas virtuais.



Estudo Complementar

http://pt.wikipedia.org/wiki/Diagrama_de_componentes

http://pt.wikipedia.org/wiki/Diagrama_de_instala%C3%A7%C3%A3o





Atividades

Com base nos sites mencionados anteriormente onde você aplicaria mais adequadamente os diagramas desta unidade?



UNIDADE 27

Objetivo: Visualizar exemplos de dois dos diagramas utilizados no mercado.

Diagrama de Objetos e Diagrama de Pacotes

Diagrama de Objetos apresenta objetos e valores de dados. Corresponde a uma instância do Diagrama de Classes, mostrando o estado de um sistema em um determinado ponto do tempo.

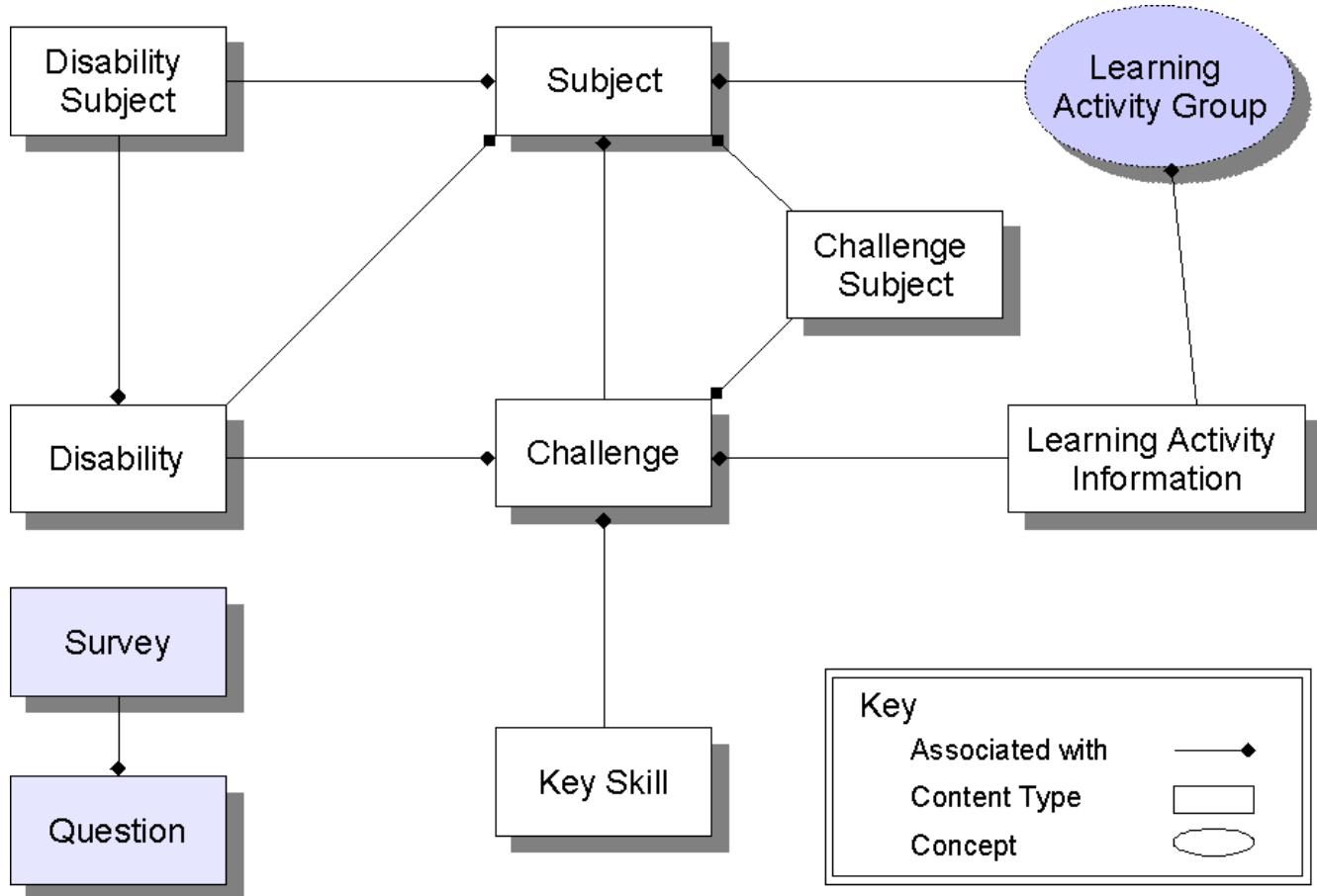
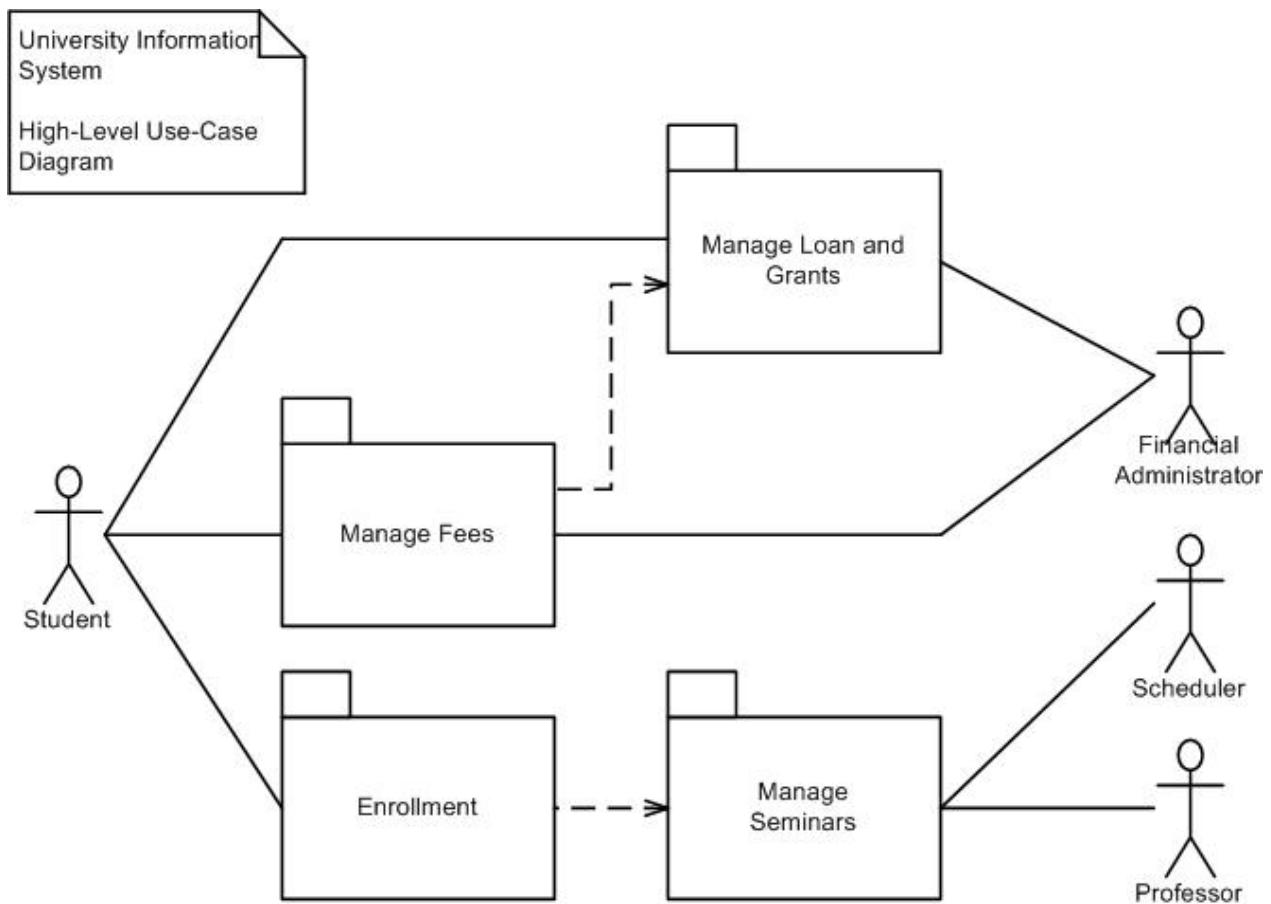


Diagrama de Pacotes é usado para organizar elementos de modelo e mostrar dependências entre eles.



Estudo Complementar

http://pt.wikipedia.org/wiki/Diagrama_de_objetos

http://pt.wikipedia.org/wiki/Diagrama_de_pacotes





Atividades

Com base nos sites mencionados anteriormente onde você aplicaria mais adequadamente os diagramas desta unidade?



UNIDADE 28

Objetivo: Através de um simples programa em Java apresentar a prática do UML

Exemplo prático: apresentação geral

Prática do UML

O primeiro programa que muitos desenvolvedores escrevem ao conhecerem uma nova linguagem é um programa simples, envolvendo apenas exibir na tela a sequência de caracteres “Hello, World !”. É um ponto de partida razoável, pois dominar essa aplicação trivial proporciona uma gratificação imediata. Além disso, manipula-se toda a infraestrutura necessária para fazer algo ser executado.

É assim que iremos mostrar a parte prática do UML. Modelar “Hello, World !” será o uso mais simples da UML que você poderá encontrar. Entretanto, essa aplicação é apenas aparentemente fácil, pois, subjacentes à aplicação, existem alguns mecanismos interessantes que a fazem funcionar. Esses mecanismos podem ser modelados facilmente com a UML, proporcionando uma visão enriquecida dessa aplicação simples.

Principais Abstrações

Vamos pegar um pequeno programa em Java para praticarmos a UML. Em Java, para o *applet* exibir “Hello, World !” em um navegador da Web é bastante simples:

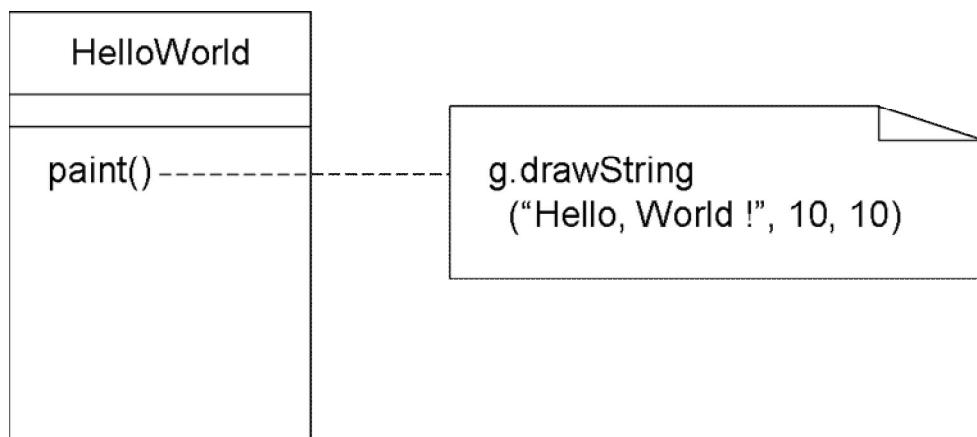
```
import java.awt.Graphics;  
  
class HelloWorld extends java.applet.Applet {  
  
    public void paint (Graphics g) {  
  
        g.drawString("Hello, World !", 10, 10);  
  
    }  
  
}
```

A primeira linha do código “`import java.awt.Graphics;`,” faz com que a classe *Graphics* fique diretamente disponível ao código indicado a seguir. O prefixo `java.awt` especifica o pacote Java de onde se importará a classe *Graphics*.

A segunda linha do código “`class HelloWorld extends java.apple.Applet {`“ introduz uma nova classe chamada *HelloWorld* e especifica que se trata de um tipo de classe semelhante a *Applet*, encontrada no pacote `java.applet`.

As últimas linhas do código declaram uma operação chamada *paint*, cuja implementação inicia outra operação, chamada *drawString*, responsável pela exibição da sequência de caracteres “**Hello, World !**” nas coordenadas fornecidas. No modo orientado a objetos usual, *drawString* é uma operação de um parâmetro chamado *g*, cujo tipo é a classe *Graphics*.

É simples a modelagem dessa aplicação na UML. Conforme mostra a figura a seguir a classe *HelloWorld* pode ser representada graficamente como um ícone retangular. A operação *paint* é mostrada nessa figura com todos os seus parâmetros formais omitidos e sua implementação especificada na nota anexa.

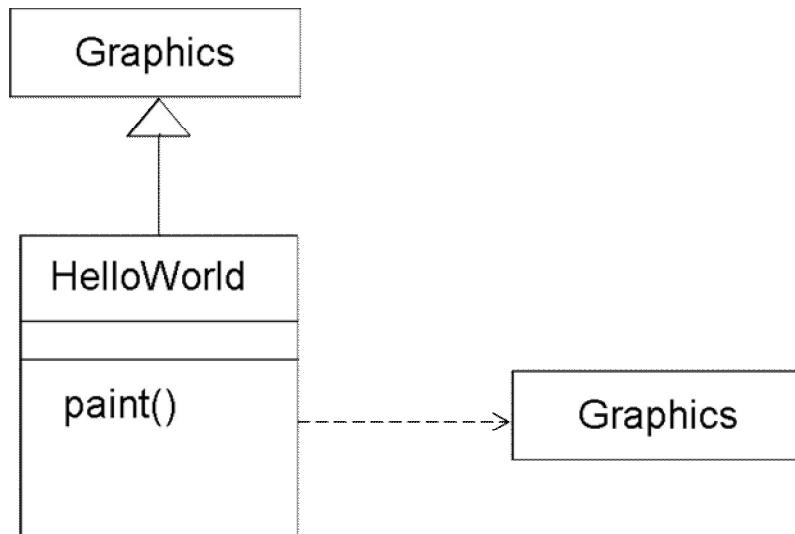


UNIDADE 29

Objetivo: Apresentar através do exemplo prático a utilização de um Diagrama de Classes

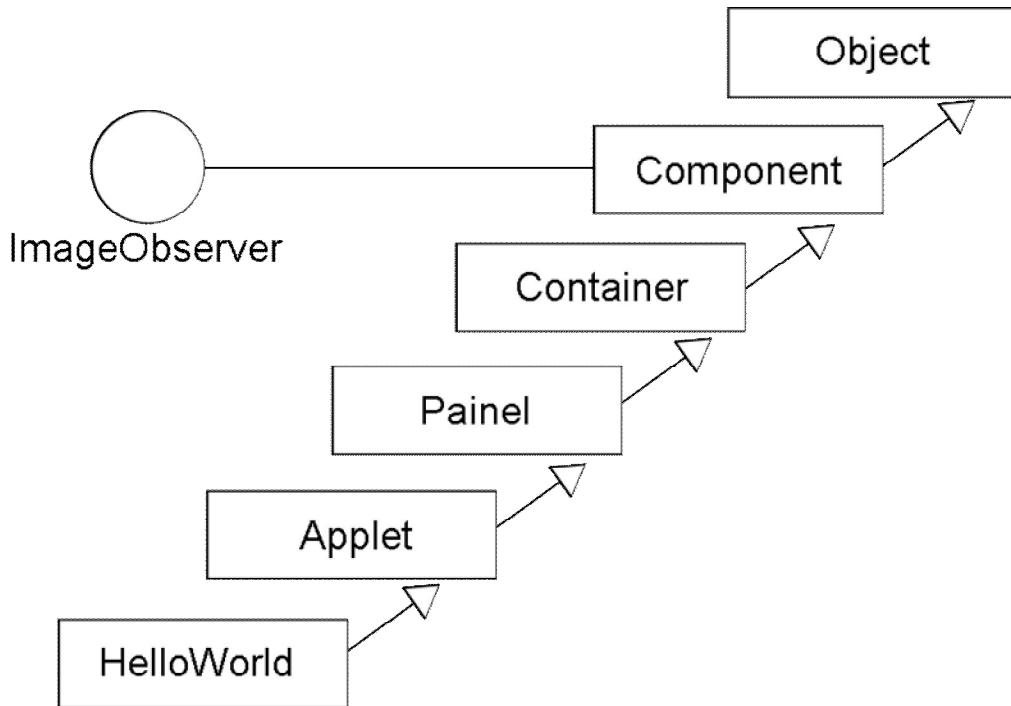
Exemplo prático: Diagrama de Classes

O **Diagrama de Classes** a seguir capta o básico da aplicação “Hello, World !”, mas não inclui vários itens. Conforme especifica o código apresentado anteriormente, duas outras classes – *Applet* e *Graphics* – estão envolvidas nessa aplicação e cada uma delas é utilizada de uma maneira diferente. A classe *Applet* é empregada como mãe de *HelloWorld* e a classe *Graphics* é usada na assinatura e implementação de uma de suas operações, *paint*. Você pode representar essas classes e seus diferentes relacionamentos com a classe *HelloWorld*, usando um **Diagrama de Classes**, conforme a figura a seguir:



As classes *Applet* e *Graphics* são representadas como ícones retangulares. As operações dessas classes não são mostradas e, portanto, seus ícones estão ocultos. A linha com a seta que vai de *HelloWorld* a *Applet* representa a generalização, significando, nesse caso, que a classe *HelloWorld* é filha de *Applet*. A linha pontilhada que vai de *HelloWorld* a *Graphics* representa um relacionamento de dependência, significando que *HelloWorld* usa a classe *Graphics*.

Esse não é o final da estrutura a partir da qual *HelloWorld* é construída. Pesquisando as bibliotecas de Java à procura de *Applet* e *Graphics*, você descobrirá que essas duas classes são parte de uma hierarquia maior. Considerando-se somente as classes que *Applet* estende e implementa, é possível gerar outro **Diagrama de Classes**, conforme mostra a figura a seguir:



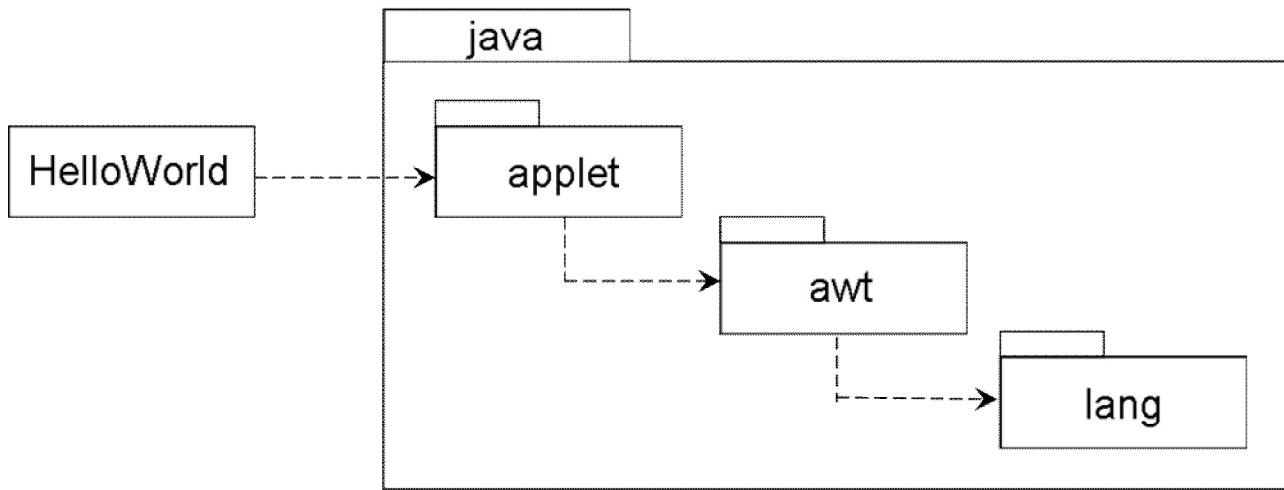
Essa figura deixa claro que *HelloWorld* é apenas uma ramificação em uma hierarquia de classes muito maior. *HelloWorld* é filha de *Applet*; *Applet* é filha de *Painel*; *Painel* é filha de *Container*; *Container* é filha de *Component*; e *Component* é filha de *Object*, que é a classe-mãe de todas as classes em Java. Portanto, esse modelo corresponde à biblioteca de Java - cada classe-filha estende a classe-mãe imediata.

O relacionamento entre *ImageObserver* e *Component* é um pouco diferente e o diagrama de classes reflete essa diferença. Na biblioteca Java, *ImageObserver* é uma interface; entre outras coisas, isso significa que não possui uma implementação e requer que outras classes a implementem. Conforme mostra a figura, uma interface pode ser representada como um círculo na UML. O fato de *Component* implementar *ImageObserver* está representado pela linha sólida que vai da implementação (*Component*) até sua interface (*ImageObserver*).

Conforme mostram essas figuras, *HelloWorld* colabora diretamente apenas com duas classes (*Applet* e *Graphics*), que, por sua vez, são apenas uma pequena parte da biblioteca maior de classes predefinidas de Java. Para gerenciar essa extensa coleção, a linguagem Java organiza suas interfaces e classes em vários pacotes diferentes.

O pacote-raiz do ambiente Java é chamado, como era de se esperar, Java. Aninhados nesse pacote existem diversos outros pacotes, contendo outros pacotes, interfaces e classes. *Object* se encontra no pacote *lang* e, portanto, seu caminho completo é *java.lang.Object*. De modo semelhante, *Painel*, *Container* e *Component* se encontram em *awt*, a classe *Applet* se encontra no pacote *applet*. A interface *ImageObserver* está no pacote *image*, que, por sua vez, pertence ao pacote *awt*, portanto, seu caminho completo é a extensa sequência *java.awt.image.ImageObserver*.

Esses pacotes podem ser visualizados em um diagrama de classes, conforme a figura a seguir:



Conforme mostra essa figura, os pacotes são representados na UML por diretórios com guias. Os pacotes podem estar aninhados, com linhas tracejadas representando as dependências entre esses pacotes. Por exemplo, *HelloWorld* depende do pacote *java.applet*, e *java.applet* depende do pacote *java.awt*.

UNIDADE 30

Objetivo: dar continuidade ao exemplo prático apresentando os mecanismos e componentes

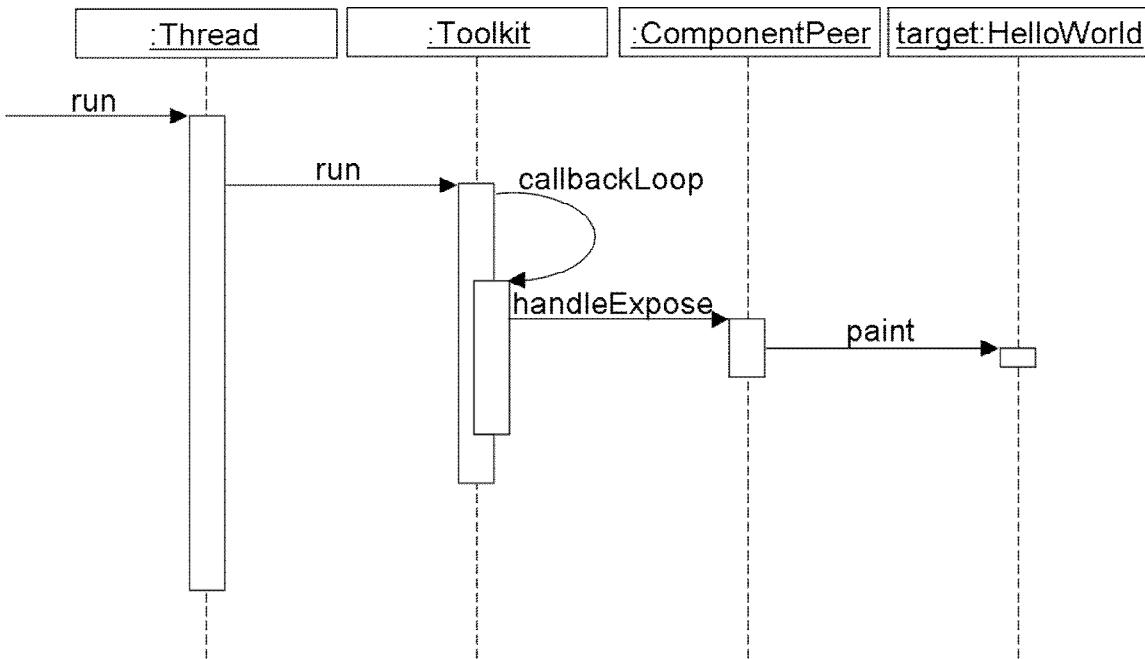
Exemplo prático: Mecanismos e Componentes

Mecanismos

A tarefa mais difícil para se dominar uma biblioteca tão rica como a da linguagem Java é aprender como suas partes trabalham em conjunto. Por exemplo, como invocar a operação *paint* de *HelloWorld*? Quais operações você deve utilizar para modificar o comportamento desse *applet*, como exibir a sequência de caracteres em outra cor? Para responder a essas e a outras perguntas, você precisará dispor de um modelo conceitual da maneira como essas classes trabalham em conjunto dinamicamente.

Um estudo da biblioteca de Java revelará que a operação *paint* de *HelloWorld* está na relação de herança de *Component*. Isso ainda mantém a pergunta de como essa operação é invocada. A resposta é que *paint* é chamada como uma parte da execução do *thread* que contém o *applet*, conforme mostra a figura mais abaixo.

Essa figura mostra a colaboração de vários objetos, incluindo uma instância da classe *HelloWorld*. Os demais objetos fazem parte do ambiente de Java e assim, na maioria dos casos, são encontrados no segundo plano dos *applets* que você criar. Na UML, as instâncias são representadas da mesma forma que as classes, mas com seus nomes sublinhados para diferenciá-las. O objeto *HelloWorld* tem um nome (*target*) conhecido pelo objeto *ComponentPeer*.

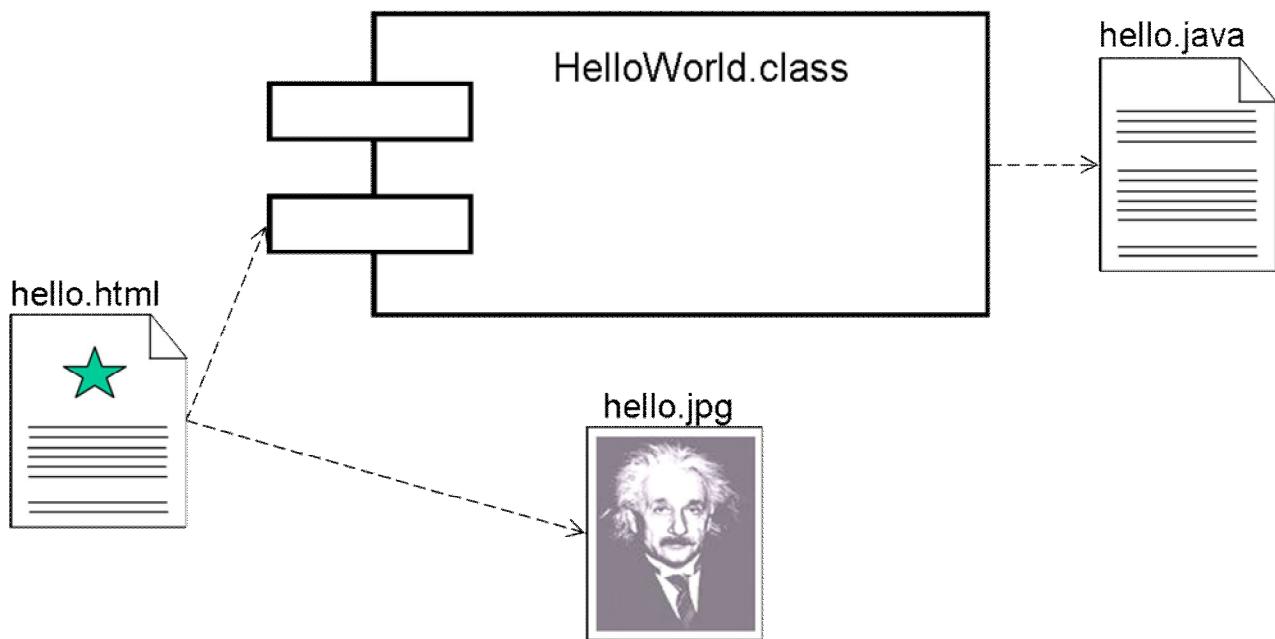


Você pode fazer a modelagem da ordenação de eventos utilizando um **Diagrama de Seqüência**, conforme mostra a figura anterior. A sequência inicia com a execução do objeto *Thread* que, por sua vez, chama a operação *run* de *Toolkit*. O objeto *Toolkit* então chama uma de suas próprias operações (*callbackLoop*) que, a seguir, chama a operação *handleExpose* de *ComponentPeer*. O objeto *ComponentPeer* então chama a operação *paint* de seu alvo. O objeto *ComponentPeer* assume que seu atributo é um *Component* (denominado *HelloWorld*) e assim a operação *paint* de *HelloWorld* é realizada polimorficamente.

Componentes

Por ser implementado como um *applet*, “**HelloWorld !**” nunca aparecerá sozinho, mas tipicamente é parte de alguma página da Web. O *applet* é executado quando a página que o contém estiver aberta, iniciado por algum mecanismo do navegador que executa o objeto *Thread* do *applet*. Entretanto, a classe *HelloWorld* não é diretamente uma parte da página da Web. Mais propriamente, é uma forma binária dessa classe, criada por um compilador Java capaz de transformar o código-fonte que representa a classe em um componente que possa ser executado. Enquanto todos os diagramas apresentados anteriormente representavam uma visão lógica do *applet*, agora estamos considerando uma visão dos componentes físicos

do *applet*. Você pode fazer a modelagem dessa visão física usando um **Diagrama de Componentes**, conforme mostra a figura a seguir:



Cada um dos ícones mostrados nessa figura representa um elemento da UML na visão da implementação do sistema. O componente chamado `hello.java` representa o código-fonte para a classe lógica `HelloWorld`; portanto, trata-se de um arquivo que pode ser manipulado pelas ferramentas de gerenciamento de configuração e ambientes de desenvolvimento. Esse código-fonte pode ser transformado no *applet* binário `hello.class` por um compilador Java, permitindo ser executado por uma máquina virtual Java em um determinado computador.

O ícone canônico para um componente é um retângulo com duas guias. O *applet* binário `HelloWorld.class` é uma variação desse símbolo básico, cujas linhas mais grossas indicam que se trata de um componente executável (assim como ocorre com uma classe ativa). O ícone para o componente `hello.java` foi substituído por um ícone definido pelo usuário, representando um arquivo de texto.

O ícone para a página da Web `hello.html` foi definido de maneira semelhante, como uma extensão da notação da UML. Conforme indica a figura, essa página da Web contém outro componente, `hello.jpg`, representado por um ícone de componente definido pelo usuário, nesse caso fornecendo uma miniatura da imagem gráfica. Como esses três últimos

componentes incluem símbolos gráficos definidos pelo usuário, seus nomes foram colocados fora dos ícones.

Observação: Os relacionamentos entre a classe (*HelloWorld*), seu código-fonte (*hello.java*) e seu código-objeto (*HelloWorld.class*), raramente são modelados explicitamente, apesar de que, de vez quando, é útil proceder dessa maneira para visualizar a configuração física do sistema. Por outro lado, é comum visualizar a organização de um sistema baseado na Web como esse, utilizando-se diagramas de componentes para a modelagem de suas páginas e de outros componentes executáveis.



Atividades

Antes de dar início à sua Prova Online é fundamental que você acesse sua SALA DE AULA e faça a Atividade 3 no “link” ATIVIDADES.



Atividades

Atividade Dissertativa

Desenvolva uma pesquisa gerando um texto, de 2 a 3 folhas adicionando imagens, de uma das unidades da nossa apostila, de sua livre escolha, permitindo a expansão da temática selecionada.

Atenção: Qualquer bloco de texto igual ou existente na internet será devolvido para o aluno realize novamente.



Glossário

Abstração A característica essencial de uma entidade que a diferencia de todos os outros tipos de entidades. Uma abstração define uma fronteira relativa à perspectiva do observador.

Ação Uma computação atômica executável que resulta em alteração do estado do sistema ou no retorno de um valor.

Ação assíncrona Uma solicitação em que o objeto emissor não faz uma pausa para aguardar os resultados.

Ação síncrona Uma solicitação em que o objeto emissor faz uma pausa para aguardar os resultados.

Adorno Detalhe da especificação de um elemento, acrescentada à sua notação gráfica básica.

Agregação Uma forma especial de associação, que especifica o relacionamento parte-todo entre o agregado (o todo) e o componente (a parte).

Agregada Uma classe que representa o “todo” em um relacionamento de agregação.

Argumento Um valor específico correspondente a um parâmetro.

Arquitetura O conjunto de decisões significativas sobre a organização de um sistema de software, a seleção de elementos estruturais e suas interfaces que compõem o sistema, juntamente com seu comportamento, conforme é especificado nas colaborações entre esses elementos, a composição desses elementos estruturais e comportamentais em subsistemas progressivamente maiores e o estilo de arquitetura que orienta essa organização — esses elementos e suas interfaces, suas colaborações e sua composição. A arquitetura de software não está relacionada somente com a estrutura e o comportamento, mas também com a utilização, funcionalidade, desempenho, flexibilidade, reutilização, abrangência, restrições e ajustes econômicos e tecnológicos e questões estéticas.

Artefato Um conjunto de informações utilizado ou produzido por um processo de desenvolvimento de software.

Assinatura O nome e os parâmetros de uma operação.

Associação Um relacionamento estrutural que descreve um conjunto de vínculos, em que o vínculo é uma conexão entre objetos; o relacionamento semântico entre dois ou mais classificadores que envolvem as conexões entre suas instâncias.

Ativação A execução de uma operação.

Associação binária Uma associação entre duas classes.

Associação enésima A associação entre três ou mais classes.

Ativar Executar a transição de um estado.

Atividade Execução não-atômica em andamento em uma máquina de estados.

Autor Um conjunto coerente de papéis que os usuários de casos de uso desempenham ao interagir com os casos de uso.

Atributo Uma propriedade nomeada de um classificador, descrevendo uma faixa de valores que as instâncias da propriedade poderão manter.

Booleano Uma enumeração cujos valores são verdadeiros ou falsos.

Característica Uma propriedade, como uma operação ou um atributo, que é encapsulada em outra entidade, como uma interface, uma classe ou um tipo de dados.

Característica comportamental Uma característica dinâmica de um elemento, como uma operação ou um método.

Característica estrutural Uma característica estática de um elemento.

Cardinalidade O número de elementos existentes em um conjunto.

Caso de uso A descrição de um conjunto de sequências de ações, incluindo variantes, que um sistema realiza, fornecendo o resultado observável do valor de um ator.

Cenário Uma sequência específica de ações que ilustram o comportamento.

Centrado na arquitetura No contexto do ciclo de vida de desenvolvimento do software, um processo que focaliza o desenvolvimento inicial e a linha de base da arquitetura de um software e então utiliza a arquitetura do sistema como um artefato primário para conceitualizar, construir, gerenciar e evoluir o sistema em desenvolvimento.

Classe A descrição de um conjunto de objetos que compartilham os mesmos atributos, operações, relacionamentos e semântica.

Classe abstrata Uma classe que não pode ser instanciada diretamente.

Classe ativa Uma classe cujas instâncias são objetos ativos.

Classe concreta Uma classe que pode ser instanciada diretamente.

Classe de associação Um elemento de modelagem que tem propriedades de classe e de associação. Uma classe de associação pode ser vista como uma associação que também tem propriedades de classe ou como uma classe que também tem propriedades de associação.

Classificação dinâmica Uma variação semântica da generalização em que um objeto poderá mudar de tipo ou de papel.

Classificação estática Uma variação semântica de uma generalização, em que um objeto poderá não alterar seu tipo, mas poderá mudar de papel.

Classificação múltipla Uma variação semântica da generalização, em que um objeto pode pertencer diretamente a mais de uma classe.

Classificador O mecanismo que descreve características estruturais e comportamentais. Os classificadores incluem classes, interfaces, tipos de dados, sinais, componentes, nós, casos de uso e subsistemas.

Cliente Um classificador que solicita serviços de outro classificador.

Colaboração Uma sociedade de papéis e outros elementos que trabalham em conjunto para proporcionar algum comportamento cooperativo maior do que a soma de todas as suas partes; a especificação de como um elemento, como casos de uso ou operações, é realizado por um conjunto de classificadores e associações desempenhando papéis específicos e utilizados de uma determinada maneira.

Comentário Uma anotação anexada a um elemento ou a uma coleção de elementos.

Componente Uma parte física e substituível de um sistema com o qual está em conformidade e proporciona a realização de um conjunto de interfaces.

Comportamento Os efeitos observáveis de um evento, incluindo seus resultados.

Composição Uma forma de agregação com propriedade bem-definida e tempo de vida coincidente das partes pelo todo; as partes com multiplicidade não fixada poderão ser criadas após a própria composição, mas, uma vez criadas, vivem e morrem com ela; essas partes também podem ser removidas explicitamente antes da morte do elemento composto.

Composta Uma classe que é relacionada a uma ou mais classes por um relacionamento de composição.

Concepção A primeira fase do ciclo de vida do desenvolvimento de um software, em que a ideia básica para o desenvolvimento é conduzida ao ponto de ser suficientemente bem-fundamentada para garantir a passagem à fase de elaboração.

Concorrência A ocorrência de duas ou mais atividades durante o mesmo intervalo de tempo. A concorrência pode ser realizada com intercalação ou executada simultaneamente por dois ou mais threads.

Condição de proteção Uma condição que precisa ser satisfeita para permitir que uma transição associada seja ativada.

Construção A terceira fase do ciclo de vida de desenvolvimento de um software, em que o software é levado da linha de base executável de uma arquitetura até o ponto em que está pronto para a transição para uma comunidade de usuários.

Container Um objeto que existe para conter outros objetos e que proporciona operações para acessar ou iterar seu conteúdo.

Contexto Um conjunto de elementos relacionados para um determinado propósito, como especificar uma operação.

Delegação A habilidade de um objeto enviar uma mensagem a outro objeto como resposta a uma mensagem.

Dependência Um relacionamento semântico entre dois itens, em que a alteração de um item (o item independente) poderá afetar a semântica do outro item (um item dependente).

Destinatário O objeto que manipula a instância de uma mensagem passada pelo objeto emissor.

Diagrama A apresentação gráfica de um conjunto de elementos, em geral representada como um gráfico conectado de vértices (itens) e arcos (relacionamentos).

Diagrama de atividades Um diagrama que mostra o fluxo de uma atividade para outra; os diagramas de atividades abrangem a visão dinâmica do sistema. Um caso especial de um diagrama de estado, em que todos ou a maioria dos estados são estados de atividades e em que todas ou a maioria das transições são ativadas pela conclusão de atividades nos estados de origem.

Diagrama de caso de uso Um diagrama que mostra um conjunto de casos de uso e atores e seus relacionamentos; o diagrama de caso de uso abrange a visão estática de caso de uso de um sistema.

Diagrama de classes O diagrama que mostra um conjunto de classes, interfaces e colaborações e seus relacionamentos. Os diagramas de classes abrangem a visão estática de projeto de um sistema; um diagrama que mostra a coleção de elementos declarativos (estáticos).

Diagrama de colaboração Um diagrama de interação que dá ênfase à organização estrutural de objetos que enviam e recebem mensagens; um diagrama que mostra as interações organizadas ao redor de instâncias e os vínculos entre elas.

Diagrama de componentes Um diagrama que mostra a organização e as dependências existentes em um conjunto de componentes; os diagramas de componentes abrangem a visão estática de implementação de um sistema.

Diagrama de gráfico de estados Um diagrama que mostra uma máquina de estados; os diagramas de gráficos de estados abrangem a visão dinâmica de um sistema.

Diagrama de implantação Um diagrama que mostra a configuração dos nós de processamento em tempo de execução e os componentes que nele existem; um diagrama de implantação abrange a visão estática de funcionamento de um sistema.

Diagrama de interação Um diagrama que mostra uma interação, composta por um conjunto de objetos e seus relacionamentos, incluindo as mensagens que podem ser trocadas entre eles; os diagramas de interação abrangem a visão dinâmica de um sistema; um termo genérico aplicado a vários tipos de diagramas que dão ênfase às interações de objetos, incluindo diagramas de colaboração, diagramas de sequências e diagramas de atividades.

Diagrama de objetos Um diagrama que mostra um conjunto de objetos e seus relacionamentos em um ponto no tempo; os diagramas de objetos abrangem a visão estática de projeto ou a visão estática de processo de um sistema.

Diagrama de seqüência Um diagrama de interação que dá ênfase à ordenação temporal de mensagens.

Domínio Uma área de conhecimento ou de atividade, caracterizada por um conjunto de conceitos e terminologia compreendidos pelos participantes dessa área.

Elaboração A segunda fase do ciclo de vida de desenvolvimento de um software, em que a visão do produto e sua arquitetura são definidos.

Elemento Um constituinte atômico de um modelo.

Elemento derivado Um elemento do modelo que pode ser computado a partir de um elemento, mas que é mostrado por questão de clareza ou é incluído com o propósito de projeto, ainda que não acrescente informações semânticas.

Elemento parametrizado O descritor de um elemento com um ou mais parâmetros não-vinculados.

Emissor O objeto que passa a instância de uma mensagem a um objeto destinatário.

Engenharia de produção O processo de transformar um modelo em código pelo mapeamento para uma linguagem específica de implementação específica.

Engenharia reversa O processo de transformação de um código em um modelo pelo mapeamento a partir de uma linguagem de implementação específica.

Enumeração Uma lista de valores nomeados, utilizada como a faixa de um determinado tipo de atributo.

Enviar A passagem da instância de uma mensagem do objeto emissor para um objeto destinatário.

Escopo O contexto que dá significado a um nome.

Espaço de nome Um escopo em que os nomes podem ser definidos e utilizados; em um espaço de nome, cada nome denota um único elemento.

Especificação Uma declaração textual da sintaxe e da semântica de um bloco de construção específico; uma descrição declarativa do que alguma coisa é ou faz.

Estado Uma condição ou situação durante a vida de um objeto, durante a qual ele satisfaz alguma condição, realiza uma atividade ou aguarda algum evento.

Estado composto Um estado formado por subestados concorrentes ou subestados em disjunção.

Estado de ação Um estado que representa a execução de uma ação atômica, tipicamente a chamada de uma operação.

Estereótipo Uma extensão do vocabulário da UML, que permite a criação de novos tipos de blocos de construção derivados dos já existentes, mas que são específicos ao seu problema.

Estímulo Uma operação ou um sinal.

Evento A especificação de uma ocorrência significativa, que tem uma localização no tempo e no espaço; no contexto de uma máquina de estados, o evento é a ocorrência de um estímulo capaz de ativar a transição de um estado.

Evento de tempo Um evento que denota o tempo decorrido desde que se entrou no estado atual.

Execução A execução de um modelo dinâmico.

Exportar No contexto dos pacotes, tornar visível um elemento fora do espaço do nome que o contém.

Expressão Uma sequência de caracteres avaliada como um valor de um determinado tipo.

Expressão booleana Uma expressão avaliada como um valor booleano.

Expressão de ação Uma expressão que é avaliada como uma coleção de ações.

Expressão de tempo Uma expressão calculada com um valor de tempo absoluto ou relativo.

Expressão de tipo Uma expressão avaliada como uma referência a um ou mais tipos.

Extremidade da associação O ponto final de uma associação, que conecta a associação a um classificador.

Extremidade do vínculo Uma instância de uma extremidade de uma associação.

Fase O intervalo de tempo entre dois marcos de progresso importantes do processo de desenvolvimento, durante o qual um conjunto bem-definido de objetivos é atingido, artefatos são concluídos e decisões são tomadas em relação à passagem para a fase seguinte.

Filha Uma subclasse.

Filho Uma subclasse.

Foco de controle Um símbolo em um diagrama de sequência, mostrando o período de tempo durante o qual um objeto realiza uma ação diretamente ou por meio de uma operação subordinada.

Fornecedor Um tipo, classe ou componente que fornece serviços que podem ser chamados por outros.

Framework Um padrão de arquitetura que fornece um template extensível para aplicações em um domínio.

Generalização Um relacionamento de especialização/generalização, em que objetos do elemento especializado (o filho) podem ser substituídos para objetos do elemento generalizado (o pai).

Herança O mecanismo pelo qual elementos mais específicos incorporam a estrutura e o comportamento de elementos mais gerais.

Herança de implementação A herança da implementação de um elemento mais específico; também inclui a herança da interface.

Herança de interface A herança da interface de um elemento mais específico; não inclui a herança da implementação.

Herança múltipla Uma variação semântica da generalização, em que um filho pode ter mais de um pai.

Herança única Uma variação semântica de uma generalização, em que um filho pode ter somente um pai.

Hierarquia de conteúdos A hierarquia de um espaço de nome, formada por elementos e os relacionamentos de agregação existentes entre eles.

Implementação Uma realização concreta do contrato declarado por uma interface; a definição de como algo é construído ou computado.

Import No contexto dos pacotes, uma dependência que mostra o pacote cujas classes poderão ser referenciadas em um determinado pacote (incluindo pacotes recursivamente nele incorporados).

Incompleto A modelagem de um elemento, em que faltam certas partes.

Inconsistente A modelagem de um elemento em que a integridade do modelo não é garantida.

Incremental No contexto do ciclo de vida do desenvolvimento de um software, é um processo que envolve a integração contínua da arquitetura do sistema para a produção de versões, cada nova versão incorporando aperfeiçoamentos incrementais em relação à anterior.

Instância Uma manifestação concreta de uma abstração; uma entidade à qual um conjunto de operações pode ser aplicado e que tem um estado para armazenar os efeitos das operações.

Integridade Como as coisas se relacionam, umas com as outras, de maneira apropriada e consistente.

Interação Um comportamento que abrange um conjunto de mensagens trocadas entre um conjunto de objetos em um determinado contexto para a realização de um propósito.

Interface Uma coleção de operações utilizadas para especificar o serviço de uma classe ou de um componente.

Iteração Um conjunto distinto de atividades com um plano de linha de base e um critério de avaliação que resulta em uma versão, interna ou externa.

Iterativo No contexto do ciclo de vida de desenvolvimento do software, um processo que envolve o gerenciamento de uma sequência de versões executáveis.

Linha de vida do objeto Uma linha em um diagrama de sequências, que representa a existência de um objeto em um período de tempo.

Localização A posição de um componente em um nó.

Mãe Uma superclasse.

Máquina de estados Um comportamento que especifica as sequências de estados pelas quais um objeto passa durante seu tempo de vida como resposta a eventos, juntamente com suas respostas a esses eventos.

Marca de tempo Uma denotação do tempo em que um evento ocorre.

Mecanismo Um projeto-padrão que é aplicado a uma sociedade de classes.

Mecanismo de extensibilidade Um dos três mecanismos (estereótipos, valores atribuídos e restrições) que permitem estender a UML de maneiras controladas.

Mensagem A especificação de uma comunicação entre objetos que contêm informações à espera de que a atividade acontecerá; o destinatário da instância de uma mensagem costuma ser considerado a instância de um evento.

Metaclasses Uma classe cujas instâncias são classes.

Método A implementação de uma operação.

Modelo Uma simplificação da realidade, criada com a finalidade de proporcionar uma melhor compreensão do sistema que está sendo gerado; uma abstração semanticamente próxima de um sistema.

Multiplicidade A especificação de uma faixa de números cardinais, que um conjunto pode assumir.

Nível de abstração Um lugar na hierarquia de abstrações, abrangendo desde os níveis mais altos de abstração (muito abstratos) até os mais baixos (muito concretos).

Nó Um elemento físico existente em tempo de execução que representa um recurso computacional, geralmente dispondo de pelo menos alguma memória e, na maioria das vezes, capacidade de processamento.

Nome O que você pode usar para denominar um item, relacionamento ou dia grama; uma sequência de caracteres utilizada para identificar um elemento.

Nota Um símbolo gráfico para a representação de restrições ou de comentários anexados a um elemento ou a uma coleção de elementos.

Object Constraint Language (OCL) Uma linguagem formal utilizada para expressar restrições secundárias de efeito livre.

Objeto Uma manifestação concreta de uma abstração; uma entidade com uma fronteira bem-definida e uma identidade que encapsula estado e comportamento; a instância de uma classe.

Objeto ativo Um objeto a que pertencem um processo ou thread e que é capaz de iniciar uma atividade de controle.

Objeto persistente Um objeto que existe depois que o processo ou o thread que o criaram deixa de existir.

Objeto transiente Um objeto que existe somente durante a execução do thread ou do processo que o criou.

Ocultar A modelagem de um elemento com determinadas partes ocultas para simplificar a exibição.

Operação A implementação de um serviço que pode ser solicitado por qualquer objeto da classe com a finalidade de afetar um comportamento.

Orientado a caso de uso No contexto do ciclo de vida do desenvolvimento de um software, um processo em que os casos de uso são utilizados como artefatos primários para o estabelecimento do comportamento desejado do sistema, para verificar e validar a arquitetura do sistema, para testar e para fazer a comunicação entre os participantes do projeto.

Orientado a riscos No contexto do ciclo de vida de desenvolvimento de software, um processo em que cada nova versão é dedicada a atacar e reduzir os riscos mais significativos para o sucesso do projeto.

Pacote Um mecanismo de propósito geral para a organização de elementos em grupos.

Padrão Uma solução comum para um problema comum em um determinado contexto.

Pai Uma superclasse.

Papel O comportamento de uma entidade que participa de um determinado contexto.

Parâmetro A especificação de uma variável que pode ser alterada, passada ou re tornada.

Parâmetro formal Um parâmetro.

Parâmetro real Uma função ou argumento de procedimento.

Pós-condição Uma restrição que precisa ser verdadeira na conclusão de uma operação.

Pré-condição Uma restrição que precisa ser verdadeira quando uma operação é chamada.

Processo Um fluxo de controle pesado, que pode ser executado concorrentemente com outros processos.

Produto Os artefatos de desenvolvimento, como modelos, código, documentação e planos de trabalho.

Projeção Um mapeamento a partir de um conjunto para um subconjunto dele.

Propriedade Um valor nomeado, denotando uma característica de um elemento.

Pseudo-estado Um vértice em uma máquina de estados que tem a forma de um estado, mas não se comporta como um estado; os pseudo-estados incluem vértices inicial, final e de histórico.

Qualificador O atributo de uma associação cujos valores dividem o conjunto de objetos relacionados a um objeto em uma associação.

Raia de natação A partição de um diagrama de interação para a organização de responsabilidades para ações.

Realização Os relacionamentos semânticos entre os classificadores, em que um classificador especifica um contrato cuja execução é garantida por outro classificador.

Receber A manipulação da instância de uma mensagem passada pelo objeto emissor.

Refinamento Um relacionamento que representa a especificação completa de algo já especificado em um determinado nível de detalhe.

Relacionamento Uma conexão semântica entre elementos.

Requisito Uma característica, propriedade ou comportamento desejado de um sistema.

Responsabilidade Um contrato ou obrigação em um tipo ou de uma classe.

Restrição Uma extensão da semântica de um elemento da UML, permitindo acrescentar novas regras ou modificar as existentes.

Restrição de tempo Uma declaração semântica sobre o valor de tempo relativo ou absoluto ou a duração.

Seqüência de caracteres Uma sequência de caracteres de texto.

Sinal A especificação de um estímulo assíncrono comunicado entre instâncias.

Sistema Possivelmente decomposto em uma coleção de subsistemas, um conjunto de elementos organizados para a realização de um propósito específico e descrito por um conjunto de modelos, provavelmente sob diferentes pontos de vista.

Solicitação A especificação de um estímulo enviado a um objeto.

Subclasse Em um relacionamento de generalização, a especialização de outra classe, a classe, mãe.

Subestado Um estado que é parte de um estado composto.

Subestado concorrente Um subestado ortogonal que pode ser mantido simultaneamente com outros subestados contidos no mesmo estado composto.

Subestado disjunto Um subestado que não pode ser mantido simultaneamente com outros subestados contidos no mesmo estado composto.

Subsistema Um agrupamento de elemento, em que alguns constituem uma especificação do comportamento oferecido pelos outros elementos contidos nesse agrupamento.

Superclasse Em um relacionamento de generalização, a generalização de outra classe, a classe filha.

Tarefa Um caminho único para execução de um programa, um modelo dinâmico ou alguma outra representação do fluxo de controle; um thread ou um processo.

Template Um elemento parametrizado.

Tempo Um valor representando um momento absoluto ou relativo.

Thread Um fluxo leve de controle que pode ser executado concorrentemente com outros threads no mesmo processo.

Tipo de dados Um tipo cujos valores não têm identidade. Os tipos de dados incluem tipos primitivos inerentes (como números e sequências de caracteres), além de tipos enumerados (como booleano).

Tipo O estereótipo de uma classe, utilizado para especificar um domínio de objetos, juntamente com as operações (mas não os métodos) que podem ser aplicadas aos objetos.

Tipo primitivo Um tipo básico, como um inteiro ou uma sequência de caracteres.

Trace Uma dependência que indica um relacionamento de processo ou de histórico entre dois elementos que representam o mesmo conceito, sem regras para derivar um a partir do outro.

Transição A quarta fase do ciclo de vida de desenvolvimento de um software, em que o software é colocado nas mãos da comunidade de usuários; um relacionamento entre dois estados indicando que um objeto no primeiro estado realizará determinadas ações e passará ao segundo estado quando um evento especificado ocorrer e as condições forem satisfeitas.

UML (Unified Modeling Language) Linguagem de Modelagem Unificada, uma linguagem para a visualização, especificação, construção e documentação de artefatos de um sistema complexo de software.

Unidade de distribuição Um conjunto de objetos ou componentes que são alocados para um nó como um grupo.

Utilização A dependência em que um elemento (o cliente) requer a presença de outro elemento (o fornecedor) para seu correto funcionamento ou implementação.

Valor Um elemento de um domínio de tipo.

Valor atribuído Uma extensão das propriedades de um elemento da UML, que permite a criação de novas informações na especificação desse elemento.

Versão Um conjunto de artefatos, relativamente completo e consistente, disponível para um usuário interno ou externo; a própria entrega desse conjunto.

Vinculação A criação de um elemento a partir de um template, fornecendo-se os argumentos para os parâmetros do template.

Vínculo Uma conexão semântica entre objetos; uma instância de uma associação.

Visão A projeção em um modelo, vista a partir de uma determinada perspectiva ou ponto de vantagem e omite as entidades que não são relevantes para essa visão.

Visão de caso de uso A visão da arquitetura de um sistema, abrangendo os casos de uso que descrevem o comportamento do sistema, conforme é visto pelos seus usuários finais, analistas e pessoal de teste.

Visão de implantação A visão da arquitetura de um sistema, abrangendo os nós que formam a topologia de hardware, em que o sistema é executado; a visão de implantação abrange a distribuição, entrega e instalação das partes que constituem o sistema físico.

Visão de implementação A visão da arquitetura de um sistema, abrangendo os componentes utilizados para montar e liberar o sistema físico; a visão de implementação inclui o gerenciamento da configuração das versões do sistema, compostas por componentes de alguma forma independentes que podem ser reunidos de vários modos para produzir um sistema em execução.

Visão de processo A visão da arquitetura de um sistema, abrangendo os threads e os processos que formam a concorrência do sistema e os mecanismos de sincronização; a visão de processo abrange o desempenho, a escalabilidade e o tempo de resposta do sistema.

Visão de projeto A visão da arquitetura de um sistema, abrangendo as classes, interfaces e colaborações que formam o vocabulário do problema e sua solução; a visão de projeto abrange os requisitos funcionais do sistema.

Visão dinâmica Um aspecto de um sistema que dá ênfase ao seu comportamento.

Visão estática Um aspecto de um sistema que dá ênfase à sua estrutura.

Visibilidade Como um nome pode ser visto e utilizado pelos outros.

R_{eferências}

BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. **UML : guia do usuário**: o mais avançado tutorial sobre Unified Modeling Language (UML). Rio de Janeiro: Campus, 2000. (tradução Fabio Freitas da Silva).

MELO, Ana Cristina. **Desenvolvendo aplicações com UML 2.0**: do conceitual à implementação. 2.ed. Rio de Janeiro: Brasport, 2004.

BEZERRA, Eduardo. **Princípios de análise e projeto de sistemas com UML**. 7. ed. Rio de Janeiro: Elsevier, 2002.

OLIVEIRA, Jayr Figueiredo de. **Metodologia para desenvolvimento de projetos de sistemas**: guia básico de referência. 2.ed. São Paulo: Érica, 1997