Universidade Federal da Paraíba
Centro de Ciências Exatas e da Natureza
Departamento de Informática

Monografia

Desenvolvimento de Sistemas Utilizando Orientação a Objetos

Soila Mara Pereira Rosado

JOÃO PESSOA – PB
julho de 2003

Universidade Federal da Paraíba
Centro de Ciências Exatas e da Natureza
Departamento de Informática

Desenvolvimento de Sistemas Utilizando Orientação a Objetos

Soila Mara Pereira Rosado

Monografia apresentada à coordenação do Curso de Especialização em Sistemas de Informação e Redes do Departamento de Informática da Universidade Federal da Paraíba – UFPB, como requisito parcial para a obtenção do grau de Especialista.

Orientador: Álvaro Francisco de Castro Medeiros- Dr.
(Orientador)

JOÃO PESSOA – PB

julho de 2003

Muitos são os planos do coração do homem,
mas é o propósito do Senhor que permanecerá.
Provérbios 19,21.

Resumo

Este documento busca transmitir uma idéia básica do desenvolvimento de sistemas orientados a objetos, não sendo aqui demonstrado nenhuma metodologia em especial. Devido à grande variedade de métodos e ferramentas, procurei manter o documento o mais genérico possível, de modo a demonstrar apenas os tópicos relevantes em uma introdução ao desenvolvimento OO. O público alvo são os analistas e desenvolvedores iniciantes na arte da OO. O assunto aqui selecionado abrange os aspectos da OO, algumas técnicas utilizadas por desenvolvedores muito experientes e introduz os conceitos mais utilizados da UML para desenvolvimento OO. Alguns padrões de desenvolvimento são apresentados como um complemento ao desenvolvimento e por se tratar de um assunto extremamente relacionado ao desenvolvimento de um sistema bem estruturado.

Sumário

Lista de Figuras

Introdução

O termo “baseados em objetos” identifica um software estruturado como uma coleção de objetos que incorporam tanto a estrutura quanto o comportamento dos dados. O oposto, que é o modelo convencional de programação, possui pouca ou nenhuma vinculação entre a estrutura e o comportamento dos dados.

A aplicação da OO durante o desenvolvimento de software ainda é bastante reduzida devido à dificuldade de encontrar profissionais capacitados para esta tarefa e à falta de credibilidade nos benefícios desta técnica.

O mercado está utilizando algumas características da orientação a objetos, mas nem tudo é aplicado, o que dificulta o aproveitamento de todas as suas vantagens. Muitas vezes, quando se desenvolve um projeto, é feita uma Análise OO mas o projeto é desenvolvido em ferramentas que não permitem que se apliquem os conceitos oferecidos, isto em geral acaba gerando dúvidas quanto aos benefícios da OO e desestimulando o profissional, fazendo com que utilize a metodologia tradicional, como a análise estruturada ou essencial e a programação convencional baseada em rotinas e eventos.

Para se obter um sistema OO é necessário que se utilize a OO durante toda a fase do ciclo de desenvolvimento, aplicando-se de forma correta as regras existentes, o que só é possível se houver o domínio da metodologia e a disponibilidade de ferramentas adequadas.

1.1 Objetivos

1.2 Objetivo Geral

Compreender as técnicas encontradas nos padrões de desenvolvimento propostos por Craig Larman, que incentivam o desenvolvimento de sistemas utilizando metodologias OO e permitindo uma visão de como é possível empregá-los na prática.

1.3 Objetivos Específicos

Apresentar um resumo da metodologia OO e da UML.
Analisar as vantagens de utilização desta metodologia durante a fase de implementação de sistemas.
Apresentar exemplos mostrando o emprego das técnicas propostas.

Orientação a Objetos

Devido à grande dificuldade de desenvolver um sistema que se adapte facilmente às mudanças nas regras de negócio que ocorrem ao longo do ciclo de vida do software, este trabalho tem como principal objetivo descrever uma visão geral dos benefícios da utilização de uma técnica de modelagem baseada em objetos, mostrando os conceitos mais utilizados e como aplicá-los, utilizando-se de uma linguagem de fácil compreensão, especialmente para aquelas pessoas que estão iniciando o estudo da OO.
Portanto, o material aqui apresentado irá parecer bastante óbvio para pessoas experientes em OO, mas certamente apresentará dicas importantes para aqueles que estão acostumados a desenvolver de forma procedural e que desejam aplicar os conceitos básicos da OO.

2.1 Histórico

A utilização de OO para o desenvolvimento de sistemas não é novidade, pois vem sendo estudada desde a década de 70. Entretanto, a prática durante todo o ciclo de desenvolvimento, isto é, da análise até a codificação, só foi aplicada a partir da década de 80 por Booch.
A partir daí muitos trabalhos foram desenvolvidos por pessoas da área que possuíam a capacidade de influenciar outros profissionais a aplicarem este novo paradigma. Se por um lado isto facilitou a divulgação das técnicas, por outro dificultou a especificação de um conceito único, pois muitos desses desenvolvedores de renome criaram suas próprias notações, dando origem a denominações diferentes para um mesmo conceito.
Após esta fase inicial de desordem, buscou-se um concenso, o que reuniu grandes conhecedores da metodologia OO com o objetivo de criar uma linguagem unificada que pudesse ser utilizada de forma coerente.
Eles criaram uma notação gráfica, símbolos padrões para a representação dos conceitos e driagramas que facilitassem o entendimento entre profissionais, o que deu origem aos conceitos de UML. Devido ao sucesso dessas técnicas, grandes empresas como Oracle, IBM, Microsoft e HP montaram um consórcio e contribuíram para a versão final de UML.

A técnica foi logo reconhecida como padrão pelo OMG(Object Managment Group), que é o órgão principal fornecedor de diretrizes para o desenvolvimento de software.

2.2 Benefícios

Agrupar os conceitos do mundo real e representá-los através de objetos é um dos maiores benefícios oferecidos pela OO, pois para nós seres humanos se torna bem mais fácil a compreensão.

Pode-se citar também como benefícios da OO:

- A Reutilização de código é considerada a grande vantagem oferecida pela OO;
- Utilização da mesma notação, durante todo o ciclo de vida do software, da análise, projeto até a implementação;[Rumbaugh]
- Facilita o reaproveitamento de código, pois temos objetos que se comportam de formas semelhantes além de permitir redução no tempo de desenvolvimento;
- Herança – torna o programa menor e facilita a manutenção. Utiliza-se as classes para implementar novas funcionalidades podendo-se herdar o comportamento (que está codificado) de outras classes anteriormente implementadas.
- Escalabilidade, que é a capacidade de uma aplicação adaptar-se facilmente às exigências de novos requisitos do sistema sem aumentar muito a sua complexidade e sem comprometer o desempenho. Para aumentar a escalabilidade a aplicação é construída utilizando-se a composição de objetos e a troca de mensagens entre estes objetos.
- O encapsulamento não permite que seja feito o acesso direto ao estado interno de um objeto, isto é, oculta e protege as informações dos objetos. O acesso aos mesmos deve ser feito através das mensagens que o objeto pode receber. O programador não necessita conhecer o código, precisa apenas da documentação dos objetos para que possa utilizá-los.
- A apropriação, que possibilita agrupar em classes os objetos com comportamento semelhante e estas classes podem ser agrupadas em hierarquias de herança , pode ser uma vantagem e também uma desvantagem. A vantagem é podermos utilizar um agrupamento de objetos e fornecermos uma solução para um problema. A desvantagem é que estes objetos serão tratados em grupos e não de forma individual.

2.3 Desvantagens da OO

A apropriação, é uma desvantagem por tratar de forma genérica diversos objetos. Isto nem sempre soluciona problemas de forma adequada, pois objetos similares são agrupados segundo uma classificação rígida. O problema é que ao ocorrer uma mudança nos requisitos do sistema pode haver a necessidade de reagrupar estes objetos, o que geralmente não é simples de ser feito. Quando se deseja classificar objetos é necessário que isto seja feito de forma precisa e inflexível, pois não é possível utilizar regras dinâmicas para classificação de objetos, isto é, para a definição das classes. A apropriação traz o problema da fragilidade nas definições, isto é outra desvantagem, pois ao ocorrer uma mudança nos relacionamentos entre as classes, pode-se ter que redefinir a hierarquia de classes. A fim de evitar tais problemas, deve-se realizar uma boa análise e esquematizar o projeto buscando evitar tais problemas, embora isto requeira maiores prazos e investimentos.

2.4 Introdução à abordagem OO

A utilização da Tecnologia OO já é uma realidade nas empresas, entretanto não está sendo largamente empregada devido geralmente à falta de profissionais com experiência no desenvolvimento das aplicações e à dificuldade de se formar estes profissionais. Além disso, nem todas as ferramentas de bancos de dados suportam todos os conceitos de OO e as que existem são extremamente caras. Empresas que utilizam a OO muitas vezes se questionam da real vantagem de sua utilização, pois devido à falta de experiência, existe dificuldade em se estimar complexidade e prazos para o desenvolvimento, o que torna a tarefa de execução mais árdua .

2.5 Principais Conceitos

A abstração de dados permite definir uma forma de representar objetos na OO. Os objetos são representados de forma abstrata para que se possa utilizá-los na solução de um problema. O comportamento destes objetos é especificado através de métodos da sua interface. A abstração de dados é um conjunto de operações e valores que especifica o comportamento de um objeto. O princípio do encapsulamento é o que permite o ocultamento das informações de um objeto e que alterações na operações desse objeto fiquem restritas à sua classe.

2.5.1 Objetos

São utilizados para representar conceitos do mundo real , ou seja informações(estruturas de dados) e a forma como serão manipuladas(procedimentos) estas informações. Um objeto é composto de propriedades, comportamento e identidade e reúne os conceitos de dados e procedimentos conjuntamente.

- Propriedades:

são as informações que representam o estado interno e atual do objeto e que somente estão acessíveis a outros objetos através de mensagens enviadas a eles.

- Comportamento: são as operações, implementadas através de métodos que atuam sobre as propriedades do objeto. Os métodos são executados ao se enviar uma mensagem solicitando a sua execução. Em geral, a mensagem possui o mesmo nome do método que ele executa. Os métodos são declarados na interface do objeto.

- Identidade

Os dados são subdivididos em entidades distintas denominadas objetos. Cada objeto tem identidade própria, mesmo que possua atributos idênticos. Estes objetos podem ser concretos(ex. um arquivo, uma bicicleta, um polígono) ou conceituais(ex. uma regra dentro de um sistema).

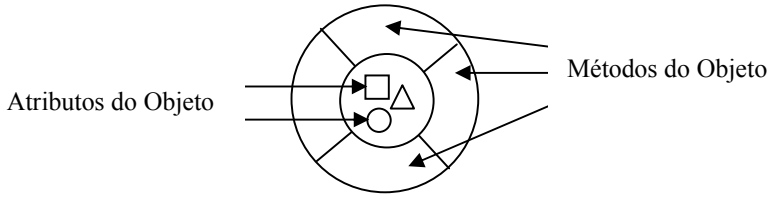


Figura 1 -Representação para um objeto

Podemos identificar com facilidade um objeto, como uma lápis, uma borracha e uma régua. Este é o conceito utilizado na OO, entretanto, também podemos considerar objeto um problema do mundo real ou uma entidade que pode ser modelada dentro do sistema e não somente os objetos conhecidos do mundo real.

A utilização de objetos traz benefícios como:

- a modularidade, pois o código deve ser escrito independentemente do fonte de outros objetos e pode-se utilizar este objeto em outros sistemas.
- o ocultamento das informações, permitindo o acesso ao objeto somente através de mensagens ou métodos declarados em sua interface pública o que facilita a sua utilização. A propriedade de ocultamento, que também pode ser chamada de abstração de dados, permite que não seja necessário o desenvolvedor saber como o objeto foi codificado, mas apenas conhecer os seus métodos da interface para poder utilizá-lo.

As mensagens enviadas aos objetos permitem a interação entre estes e aumentam a funcionalidade, permitindo implementar comportamentos mais complexos. A mensagem possui três componentes básicos:

- o objeto ao qual se deseja enviar a mensagem, que é chamado receptor;
- o nome do método que será executado;
- os parâmetros(se existirem) necessários para que o método seja executado.

Exemplo:

Um objeto A deseja executar um método de B. Neste caso, o objeto A envia uma mensagem para o objeto B, chamando o nome do método e passando os parâmetros necessários para execução do mesmo.

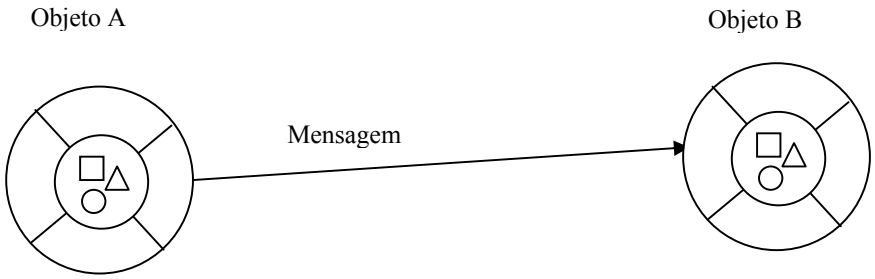


Figura 2 – Representação para a passagem de mensagens entre objetos.

2.5.1.1 Métodos

Os métodos implementam o comportamento de um objeto. O comportamento é o modo como um objeto responde a uma mensagem recebida. O método é uma função ou procedimento que é definido em uma classe de objetos e é usada para realizar operações que alteram o estado interno de um objeto.

As operações de uma classe são os métodos implementados na linguagem escolhida, podendo haver diversos métodos para uma mesma operação. Ao chamar o método, a linguagem de programação baseada em objetos o faz automaticamente baseando-se no nome da operação e na classe do objeto que está sendo utilizado para realizar a chamada.

Ex.: Método ReservarLivro(objeto:Livro);

Poderemos ter no objeto Livro um método ReservarLivro com os parâmetros código do Livro e data da reserva, que seria chamado da seguinte forma:

Livro.ReservarLivro(10,07/10/2002);

Para se criar um objeto é preciso termos um método construtor e para destruí-lo , um método destrutor.

Os construtores criam e inicializam uma nova instância de um objeto, isto é, alocam memória para o mesmo.

Exemplo1: Considerando-se que temos definido anteriormente uma classe TLivro, o construtor para criar um novo livro poderá ser chamado da seguinte forma :

Livro := TLivro.Create(application);

O destrutor libera a área de memória que foi alocada no ato da criação do objeto.

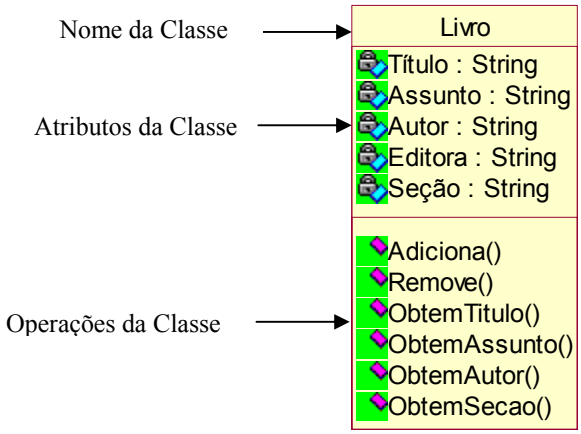
Exemplo2 : Livro.Free;

2.5.2 Classes

Os objetos com a mesma estrutura de dados(atributos) e o mesmo comportamento(operações) são agrupados em uma classe. A classe descreve um conjunto infinito de objetos. Cada objeto é uma instância de sua classe e possui valor próprio para cada atributo.

O conceito de classe serve para identificar objetos semelhantes, por isso pode-se dizer que este conceito é relativo ao que você deseja considerar como uma classe. Pode-se dizer que Livro, carro, cavalo, computador fazem parte da classe Patrimônio, se estivéssemos pensando em termos de bens com valores financeiros. Entretanto, estes mesmos objetos podem ser agrupados em classes diferentes como Livros,Imóveis, Animais, Automóveis , Equipamentos.

Representação gráfica da classe Livro



Atributos : Título, Autor, Assunto, Editora, Seção

Operações : Adiciona(), Remove(),ObtemAutor.

Figura 3 - Representação de uma classe

Podemos considerar que as classes são objetos, mas como não são objetos do mundo real, devemos chamá-las de metaobjetos. Os metaobjetos descrevem objetos, que possuem suas próprias classes, que são descritas por suas metaclasses.

As classes possuem atributos, que armazenam informações para novos objetos criados a partir de uma classe e possuem um valor comum para toda uma classe de objetos.

Autor, Assunto, Editora, Preço são atributos da classe Livro.

As operações de uma classe são funções que são executadas sobre ela própria. As instâncias de uma classe herdam todas as operações da classe.

Podemos citar como exemplo a criação de uma nova instância da classe, que necessariamente deve ser uma operação da classe a partir da qual será criada uma nova instância. Esta operação não poderia ser da nova classe instanciada pois o objeto ainda não foi criado.
Podemos ter uma mesma operação que se aplica a classes diferentes. Este tipo é conhecida como operação polimórfica, pois toma formas diferentes em classes diferentes.
As operações de uma classe são implementadas através dos métodos. No exemplo acima, temos a operação ObterDadosdaPessoa, que será um método da classe Pessoa. Esta operação poderá ser implementada com o mesmo nome nas classes usuário e funcionário, que serão subclasses da classe Pessoa, entretanto, terão formatos diferentes em cada uma dessas classes.

Classes Concretas e Classes Abstratas
Uma classe abstrata não poderá jamais ser instanciada diretamente, somente suas subclasses, desde que estas sejam concretas. As classes abstratas são utilizadas para encapsular classes que participam da mesma associação ou agregação e para definir métodos que serão herdados pelas subclasses, Uma classe abstrata possui operações abstratas, que são métodos que serão implementados nas subclasses concretas.

As classes concretas são instanciáveis diretamente. A classe Pessoa pode ser implementada como uma classe abstrata. Classes concretas não possuem operações abstratas, entretanto, a classe poderá ser refinada e se tornar uma classe abstrata, bastando que para isto acrescentemos um método abstrato na subclasse. Uma classe abstrata poderá ser refinada em uma classe concreta, desde que sua subclasse não possua métodos abstratos.
As classes são descritas graficamente em um diagrama de classes.

2.5.3 Herança
É o compartilhamento de atributos e operações entre classes baseado em um relacionamento hierárquico. A herança é o mecanismo da OO que possibilita a

reutilização das propriedades de uma classe na definição de outras classes e permite também a construção de sistemas reutilizando componentes existentes.
Podemos definirmos uma classe, chamada subclasse, que herda a definição de uma classe pré-existente, chamada superclasse. A subclasse herdará todas as operações da superclasse, podendo também ser implementados outros métodos para esta nova classe.
Podemos exemplificar a herança da seguinte maneira:
A subclasse usuário herda as características da superclasse Pessoa. Neste caso, a classe usuário herdará todo o comportamento e o estado de sua superclasse. Na classe usuário pode-se incluir novos métodos ou mesmo redefinir métodos da superclasse Pessoa.
Os métodos que forem adicionados irão tornar a subclasse cada vez mais especializada. As alterações que ocorrerem na superclasse irão se refletir na subclasse, conseqüentemente, poderão comprometer a implementação de suas subclasses.
Existem linguagens OO que permitem a herança múltipla que consiste em fazer com que uma subclasse herde características de mais de uma superclasse.

Ex.: Define-se uma classe geral Pessoa e posteriormente refina-se esta classe em subclasses como usuário, Professor, Funcionário. A classe Pessoa possui operações genéricas como InserirNovaPessoa(), AlterarDadosPessoa(), ObterDadosPessoa() . A classe usuário herda todas os atributos e operações da classe Pessoa, porém pode-se acrescentar outras operações e outros atributos que serão somente da classe usuário e das subclasses dela derivada, como exemplo CadastrarNovoUsuário (). Cada objeto conhece sua classe através da herança.

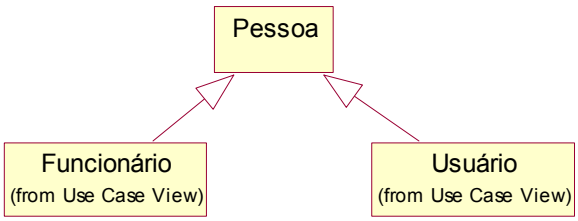


Figura 4 – Representação de Herança

2.5.4 Polimorfismo

A aplicação do polimorfismo torna o código mais legível, mais enxuto e facilita a manutenção dos sistemas pois permite que se utilize métodos com o mesmo nome para objetos diferentes. A mesma operação, que foi implementada através da codificação de um método, pode atuar de modo diferente em classes diferentes. Isto significa que objetos diferentes podem responder a uma mesma mensagem de forma diferente.

Ex.: Ao tentarmos inserir uma nova pessoa, precisaríamos de três método diferentes, um para usuário, um para Professor e um método para Funcionário.

```
If Pessoa then          {método}
   If Pessoa = Professor then
        Pessoa.InsereProfessor
    else
       if Pessoa = usuário then
          Usuário.InsereUsuário
       else
         if Pessoa = Funcionário then
            Funcionário.InsereFuncionário

Professor.InsereProfessor;   {chamada do método}

Usuário.InsereUsuario;
Funcionário.InsereFuncionario;
```

Professor, Usuário e Funcionário são instâncias de objetos das respectivas subclasses Professor, Usuário e Funcionário.

Utilizando-se o polimorfismo, teremos três classes e apenas um método Insere:

```
Pessoa.Insere;  {método}
Professor.Insere;{chamada do método}
Usuário.Insere;
Funcionário.Insere;
```

O exemplo mostra que a operação Insere pode atuar de forma diferente em objetos diferentes. A utilização do polimorfismo não facilita a reutilização de código, mas diminui o número de linhas de forma significativa quando temos muitos objetos com comportamentos ligeiramente diferentes. Este conceito não exige que se conheça a implementação do método insere para cada objeto.

O tipo de polimorfismo acima é um exemplo de métodos com implementação diferente com o mesmo nome em classes diferentes. Existe outro tipo de polimorfismo, chamado polimorfismo paramétrico ou sobrecarga de operadores, onde podemos ter métodos com o mesmo nome, implementados na mesma classe, mas que diferem apenas pelo número de parâmetros que utilizam.

Algumas linguagens OO implementam o polimorfismo com ligação(binding) dinâmica ou com ligação estática.

A ligação dinâmica é feita em tempo de execução e poderá ser alterada durante a execução do programa, entretanto, esta facilidade torna a execução do sistema mais lenta.

A ligação estática é feita em tempo de compilação ou interligação (linking) não podendo mais ser alterada no decorrer da execução do programa.

Técnicas de Modelagem Orientadas a Objetos

3.1 Histórico

A necessidade de uma técnica para desenvolvimento de sistemas OO surgiu à medida que as linguagens de programação OO também se desenvolviam. A linguagem inicial OO foi a Simula, utilizada na década de 80. A partir daí surgiram outras linguagens, como C++ e Smaltalk. Muitos métodos foram criados para dar apoio ao desenvolvimento OO:

- O método Booch – Criado por Grady Booch,permitia que o sistema fosse analisado utilizando-se uma série de diagramas, que eram construídos através de diferentes visões do sistema. Estas visões eram construídas através de micro e macro análises dos processos do sistema, utilizando para isso um processo interativo. Por ser um método muito extenso, não se tornou amplamente utilizado.

- TMO - A Técnica de Modelagem a Objetos (OMT - Object Modeling Tecnique), é um método desenvolvido na General Eletric onde James Rumbaugh (um dos criadores da UML), havia trabalhado. Era um processo voltado para testes. O sistema era descrito por vários modelos: o modelo para objetos, o modelo dinâmico, o modelo funcional, e o modelo use-case. Cada modelo complementava o outro dando uma descrição completa do sistema. O método OMT também era muito prático nas descrições de como fazer um desenho de sistema, levando em conta a ocorrência e as relações entre os sistemas;

- OOSE/Objectory: formulado por Ivar Jacbson ,os métodos OOSE e Objectory foram construídos sob o mesmo ponto de vista básico. O método OOSE é uma visão de Jacobson de um método Orientado a Objetos. O método Objectory é usado para construção de vários sistemas (como o das telecomunicações para a Ericsson, ou financeiro para as empresas de Wall Street). Ambos os métodos se baseiam no uso de caixas, que define as características principais do sistema vistas pelo ator externo. Essas caixas são usadas em todas as fases do desenvolvimento do sistema;

- Fusão: O método de fusão vem da Hewlett-Packard. Foi chamada de segunda geração de métodos porque é baseada nas experiências de muitos dos métodos iniciais. Herdou uma série de importantes idéias usadas já anteriormente. Incluiu técnicas para a especificação das operações e interações entre os objetos. Este método usa um grande número de modelos de diagramas;

- Coad/Yourdon: Este método, também conhecido como OOA/OOD foi um dos primeiros métodos usados na análise e projeção da orientação a objetos. O método era preferível por ser simples e facilmente compreensível e porque trabalhava bem a terminologia da linguagem orientada a objetos. Contudo, a notação e o método eram apenas úteis em sistemas limitados. Conseqüentemente, é muito pouco usado.

Cada um desses métodos tinha sua própria notação (os símbolos e desenhos das orientações), processos (as atividades eram executadas em diferentes partes do desenvolvimento), e ferramentas (a caixa de ferramentas que dava suporte e notação para o processo). Isso tudo fazia da escolha do método mais adequado uma decisão muito importante, e freqüentemente levava a longas discussões sobre qual método era o melhor, o mais avançado e o correto para cada projeto específico. Não chegando a ponto nenhum, já que cada método tinha seus pontos fortes e fracos, desenvolvedores experientes escolhiam um método e adaptavam boas idéias de outros.
Para evitar esses problemas é que foi criada a UML.

3.2 Diferenças em Relação à Análise estruturada

A análise estruturada baseia-se em processos e funções (Yordon). A ênfase é na especificação e decomposição da funcionalidade do sistema. Caso os requisitos mudem pode ser necessária uma grande reestruturação no sistema. Na análise OO, procuramos decompor o problema identificando os conceitos que pertencem ao domínio e documentar estes conceitos em um modelo conceitual.
A TMO busca primeiro identificar os objetos contidos no domínio da aplicação e depois em estabelecer os procedimentos relativos a eles. Durante a programação, a herança da estrutura de dados e do seu comportamento permite o compartilhamento por diversas subclasses sem redundâncias. Logo, o compartilhamento de código devido à utilização da herança representa uma das principais vantagens das linguagens baseadas em objetos.

UML

4.1 Histórico

O desenvolvimento da UML - Unified Modeling Language teve início em 1994 pelos renomados desenvolvedores Grady Booch e James Rumbaugh, da Rational Software Corporation. Eles buscavam unir os seus métodos(Booch e OMT2) com objetivo principal de criar um método novo, o "Método Unificado" ("Unified Method"). Em 1995, Ivar Jacobson (desenvolvedor dos métodos OOSE e Objectory), uniu-se a eles. A Rational Software comprou a Objective Systems, companhia que distribuía o Objectory, e então os três principais pesquisadores empenharam-se na criação do que se tornaria conhecido como "Unified Modeling Language" , a linguagem de modelagem unificada - UML.

Surgiram diversas versões de teste da UML para utilização dos desenvolvedores que faziam uso da OO. A partir daí, muitas foram as sugestões para melhorar a linguagem. A versão 1.0 da Uml foi oficialmente lançada em Janeiro de 1997.

A UML é uma linguagem padrão de modelagem, ou seja, uma linguagem cujo vocabulário e regras têm seu foco voltado para a representação conceitual e física de um sistema. Entretanto, nenhum modelo é inteiramente suficiente. Sempre são necessários vários modelos, conectados entre si, para tornar possível entender qualquer aspecto, ainda que o sistema seja simples. No caso de sistemas que fazem uso intenso de software, torna-se essencial uma linguagem capaz de abranger as diferentes visões relacionadas à arquitetura de linguagem e de como essa arquitetura evolui ao longo do ciclo de vida de desenvolvimento do software.

A UML é uma linguagem para visualização, especificação, construção e documentação de um sistema complexo.

4.2 O que é a UML

Inicialmente, é necessário entender as diferenças entre Método e Linguagem de Modelagem. Um método não é o mesmo que a linguagem de modelagem. Um método é uma forma, uma regra para a estruturação de pensamentos e ações. Um método diz ao usuário o que fazer, como fazer, quando fazer, e por que isso é feito.Os métodos dispõem de modelos que são usados para descrever algo e para mostrar os resultados do método utilizado. Um método difere de uma linguagem de modelagem principalmente porque a linguagem, como é o caso da UML, não possui um processo ou instruções para o que fazer, como fazer, quando fazer e por que algo é feito.

A estruturação de nossos pensamentos é demonstrada através da construção de um modelo, que possui um propósito para que se saiba como utilizá-lo. Entretanto, para expressarmos um modelo fazemos uso de uma linguagem de modelagem, que é uma notação, que possui símbolos para a construção do modelo desejado. Além disso são especificadas regras sintáticas, semânticas e pragmáticas que definem como usar estes símbolos.

As regras sintáticas esclarecem como os símbolos devem ser e como combiná-los na linguagem de modelagem. A sintaxe define como agrupar, em que seqüência e de que forma usar estes símbolos. Comparando à linguagem humana, é como falar, como formar as frases, ou seja, como agrupar estas palavras para dar sentido à uma oração. As regras semânticas indicam o significado de cada símbolo e como interpretá-lo. As intenções dos símbolos são definidas pelas regras pragmáticas para que sejam compreensíveis às pessoas.

O entendimento dessas regras nos permite utilizarmos bem uma linguagem de modelagem. Podemos comparar um escritor que deseja descrever uma estória em linguagem comum, sendo esta apenas uma ferramenta que o escritor deve dominar e ficando a cargo do autor escrever uma boa história.

Portanto, devemos lembrar que a UML não é um método de desenvolvimento de sistemas, mas apenas uma linguagem que pode e deve ser usada em todas as suas fases.

4.3Visões

Mostram os diferentes aspectos do sistema que está sendo modelado, consiste em uma abstração apoiada em Diagramas. Ao definirmos as visões, especificamos os aspectos particulares do sistema que cada uma mostrará, enfocando diferentes níveis de abstração e então teremos uma figura completa do sistema. Podem existir alguns casos de sobreposição entre os diagramas, o que significa que um destes pode fazer parte de mais de uma visão. Os diagramas compõem a visão dos modelos de elementos do sistema.

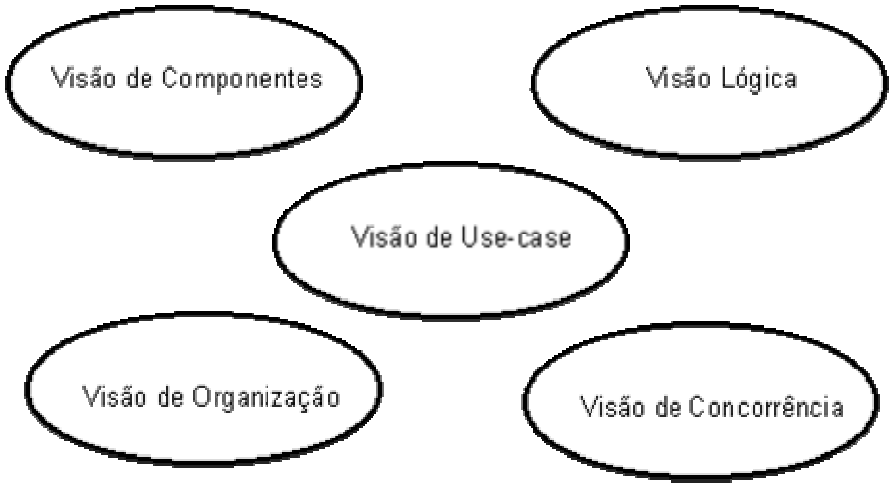


Figura 5 – Diferentes Visões do Sistema

As visões também podem servir de ligação entre a Linguagem de Modelagem e o método de desenvolvimento escolhido. As visões são complementares entre si. Por exemplo, um caso de uso envolverá a colaboração de algumas classes que foram definidas. Determinado agrupamento de classes será empacotado em um único componente – programa fonte. Um conjunto de programas fontes serão compilados e darão origem a um programa executável que, por sua vez, será executado em determinado nó da rede. Todas essas informações reunidas especificam a arquitetura do sistema.

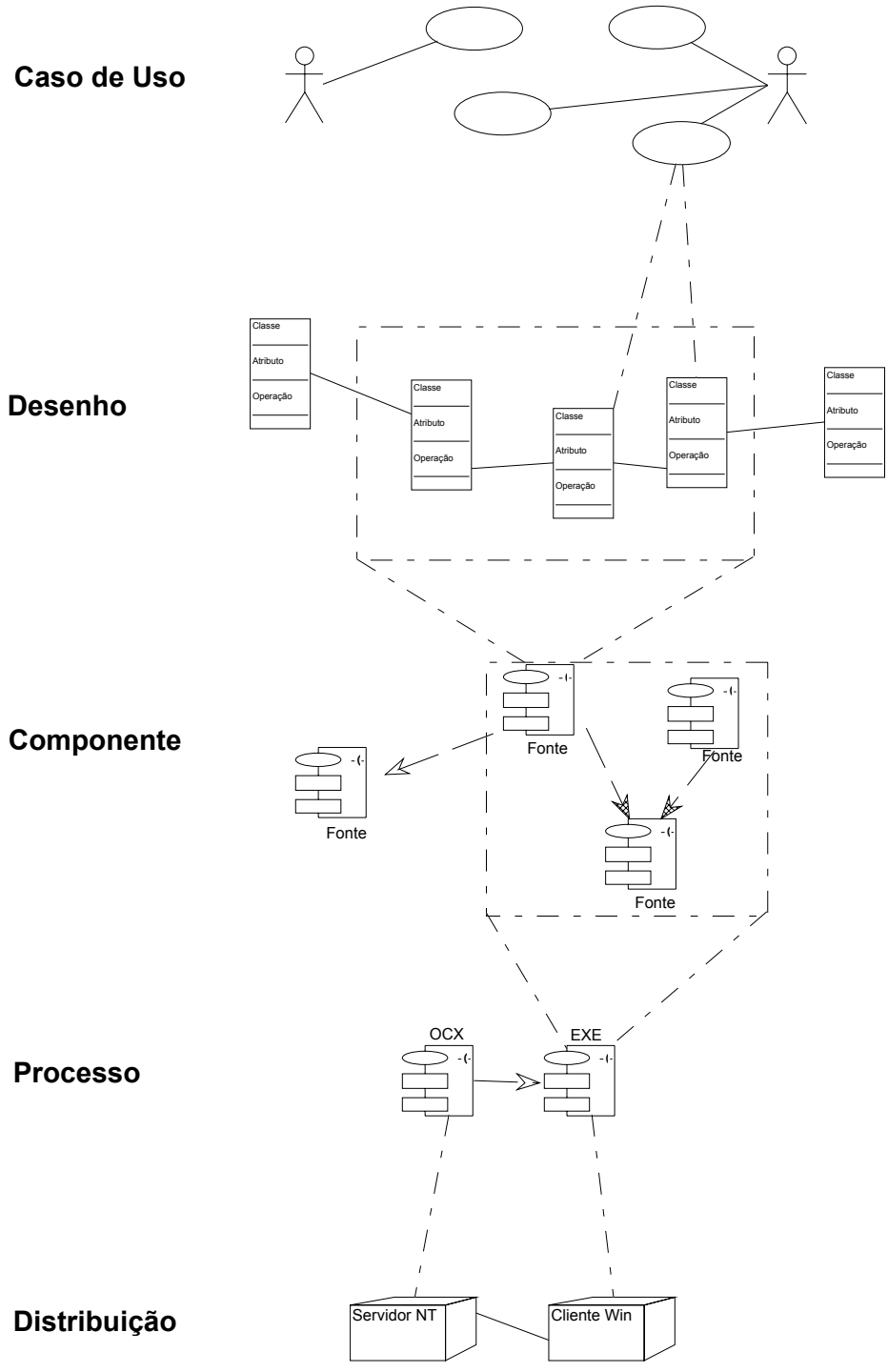


Figura 6 – Representação de uma visão do sistema.

4.4 Modelo de Elemento

São modelos estruturais, isto é, são os substantivos utilizados em modelos da UML. Atualmente, a UML possui diversos modelos, sendo alguns deles : classes, associações, generalizações, relacionamentos, interface, colaborações, casos de uso, componentes, nós, classes, objetos, estados, pacotes. Cada modelo de elemento possui uma representação gráfica distinta. Um mesmo elemento pode existir em diversos tipos de diagramas, porém , existem regras para definir que elementos poderão ser mostrados em cada tipo de diagrama. Os relacionamentos também são modelos de elementos e são usados para conectar outros modelos de elementos entre si.

4.4.1 Classes

Um software OO utiliza-se do conceito de classes de objetos e não de estruturas procedurais. As classes são conjuntos de objetos com as mesmas propriedades(atributos), apresentam o mesmo tipo de comportamento(operações) e relacionam-se(associações) da mesma maneira com outros objetos. Uma classe é a descrição de um tipo de objeto. Todos os objetos são instâncias de classes, onde a classe descreve as propriedades e comportamentos daquele objeto. Objetos só podem ser instanciados de classes. As classes são representadas no Diagrama de Classes, que também mostram atributos e operações de uma classe e as restrições no que se refere à comunicação com outros objetos. O Diagrama de classes salienta definições para classes de software e de interfaces de uma aplicação.

Os métodos de uma classe implementam a sua funcionalidade. São usados para enviar mensagens para outras classes do diagrama.

Exemplo: classe Pessoa

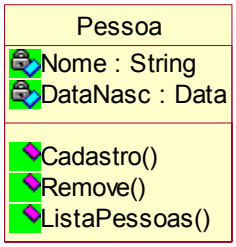


Figura 7 – Representação de uma classe e seus métodos.

Objetos

A UML utiliza-se do conceito de OO para definir um objeto e sua representação é feita através de uma classe. Deve-se usar o nome do objeto sublinhado, para representar uma instância da classe.

Estado

O estado é uma propriedade de um objeto que representa a condição atual deste objeto. O estado de um objeto é geralmente alterado por eventos que executam operações no objeto e que alteram os seus atributos. O estudo dos estados de um objeto permite que se determine os comportamentos possíveis do mesmo. Podemos representar para uma classe de objetos um diagrama de Estados. Este tipo de diagrama demonstra o comportamento de uma única classe de objetos, visto que os objetos de uma classe possuem o mesmo comportamento.

4.4.2 Casos de Uso

Os casos de uso são utilizados para identificar os requisitos funcionais que o sistema deve atender, mas não especificam explícitamente estes requisitos. A finalidade ao se especificar um caso de uso é melhorar a compreensão dos requisitos do sistema, pois é um documento feito em forma de narrativa que descreve os processos do domínio.

Um caso de uso descreve a sequência de eventos executados por um ator, isto é, processos do negócio que representam os casos de utilização de um sistema efetuados por um agente externo(o ator, que é o usuário do sistema). Os Modelos de casos de uso irão conter os diagramas de caso de uso do sistema.

Os casos de uso são agentes externos ao sistema e não se deve esperar correlação entre eles e as classes do sistema. Os casos de uso não representam passos individuais nem operações ou transações. Não podemos ter um caso de uso que represente uma sequência como Imprimir comprovante de Empréstimo, Pagar Atraso de Devolução. O caso de uso descreve uma seqüência do início ao fim de um grande processo, que possui muitos passos ou transações. Podemos, entretanto, dividir um caso de uso extenso em subcasos, os quais serão casos de uso abstratos, podendo estes chegarem a descrever passos individuais, porém isto não é a regra.

Para a identificação dos casos de uso podemos fazer uso de brainstorming e revisões de documentos dos requisitos do sistema. O ideal é começarmos inicialmente identificando

os atores e para cada ator identificar os processos que são iniciados por ele ou dos quais o ator participa. Outra forma de identificar os casos de uso é identificar os eventos externos aos quais o sistema deve responder e relacioná-los a atores e a casos de uso.

Atores

Um ator é um tipo (uma classe), e não uma instância.. Um caso de uso é sempre inicializado por uma mensagem enviada por um ator, o que chamamos de estímulo.

Atores são entidades externas ao sistema que o estimulam com eventos de entrada ou que recebem algo do mesmo. Atores são descritos de acordo com o papel desempenhado em relação ao sistema. Um mesmo ator poderá desempenhar diversos casos de uso. Um ator pode ser um ser humano, um sistema externo que utilize informações do sistema atual. Quando um caso de uso é executado, ele envia mensagens para um ou mais atores.

Precisamos identificar o que é externo e o que é interno a um sistema, além de especificar as suas funcionalidades. Podemos dizer que as fronteiras do sistema são especificadas pelo software/ hardware de um dispositivo ou sistema de computador, um departamento ou organização. Os atores representam somente o ambiente externo do sistema. Dessa forma, a definição do que é externo ou não ao sistema é relativa. Se escolhermos a biblioteca como a fronteira do sistema, então o funcionário está interno ao sistema e o usuário será um recurso externo do sistema, isto é, um ator. Se escolhermos o software como sistema, então funcionário e Usuário serão agentes externos. Usaremos esta última opção para representar os exemplos.

Atores também podem ser definidos como ativos ou passivos. Um ator ativo é aquele inicializa o estudo de caso, enquanto que o passivo nunca inicia um estudo de caso, mas apenas participa de outros usos de caso. Quando se identifica um ator, está se estabelecendo uma entidade interessada em usar e interagir com o sistema.. Apesar do ator não ter inicializado o caso de uso, o ator irá em algum momento se comunicar com um. O ator recebe um nome que reflita seu papel no sistema.

Em UML, existem classes do tipo <<ator>>, e o nome da classe é o nome do ator (como dito acima, refletindo seu papel), uma classe ator, como outra qualquer, possui atributos e métodos, bem como documentação própria descrevendo o ator.

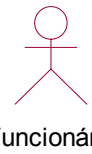


Figura 8 - Representação de Ator.

A UML não especifica um formato padrão para a descrição de casos de uso, entretanto, no decorrer do uso destes casos de uso surgiu uma estrutura padrão como é mostrada a seguir.

Inicialmente apresentamos informações resumidas, seguindo o formato abaixo:

Caso de Uso: Nome do caso de uso

O nome do caso de uso deve iniciar com um verbo – Reservar Livro – para enfatizar que é um processo.

Atore(s) : Lista de atores, indicando qual ator inicia o caso de uso

Finalidade : Objetivo do caso de uso

Visão Geral: descrição resumida do caso de uso

Tipo : primário ou essencial

Exemplo : Um usuário solicita o empréstimo de um livro, o funcionário irá efetuar a tarefa de realizar empréstimo para o usuário.

Exemplo1 : Funcionário: Registrar Empréstimo

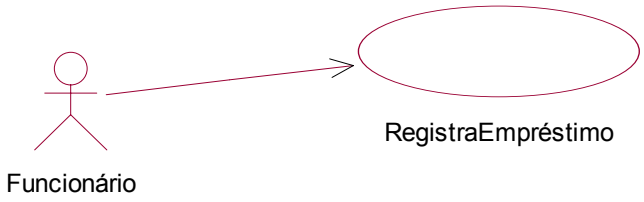


Figura 9 - Representação do Caso de Uso Registrar Empréstimo

Descrição do caso de uso Registrar Empréstimo

Seqüência Típica de Eventos

Caso de Uso : Registrar Empréstimo

Ações do Ator	**Resposta do Sistema**
1. O funcionário recebe do usuário livro(s) que deseja levar emprestado.	
2. O funcionário registra no sistema os dados do livro para verificar se é possível realizar o empréstimo	**3.** O sistema emite dados identificando que o livro é para empréstimo (ou para consulta).
4. Considerando que o livro é para empréstimo, o funcionário registra os dados do usuário.	**5.** O sistema emite confirmação de que o usuário está apto a obter o empréstimo.
6. O funcionário confirma o empréstimo.	**7.** O sistema registra os dados e emite comprovante de empréstimo para o usuário com dados do usuário, dados do livro e a data da devolução.

4.4.3 Componentes

São partes físicas e substituíveis de um sistema, que proporcionam a realização de um conjunto de interfaces. Em um sistema, encontram-se diferentes tipos de componentes que são artefatos do processo de desenvolvimento, como os arquivos de código-fonte e dlls. Tipicamente, representam o pacote físico de elementos lógicos diferentes, como classes, interfaces e colaborações. Graficamente são representados como retângulos com abas.

Figura 10 – Representação de Componente.

4.4.4 Colaborações

Definem as iterações, o comportamento cooperativo resultado da soma das funções dos elementos. Assim, as colaborações contêm dimensões estruturais e comportamentais. Graficamente são representadas como elipses tracejadas com seu nome.

4.4.5 Pacote

O pacote é utilizado para a agrupar elementos que estão relacionados. Um pacote é um componente puramente conceitual, ou seja, só existe em tempo de execução. Graficamente é representado como diretórios com guias. O pacote é proprietário de elementos como tipos e classes. Para um modelo conceitual devemos fazer com que todos os seus elementos pertençam a um mesmo pacote. Podemos, por exemplo, definir pacotes que agrupem funções de uso geral dentro do sistema e chamá-lo Geral.

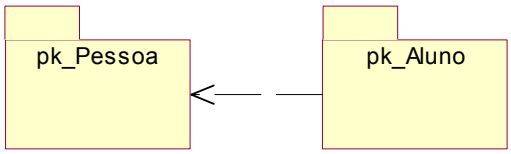


Figura 11 – Representação de Pacote.

4.4.6 Interface

É uma coleção de operações que especificam serviços de uma classe ou componente. Ou seja, uma interface representa todo o comportamento externamente visível do elemento. Pode representar todo o comportamento de uma classe ou componente, ou apensa parte desse comportamento. A interface define um conjunto de especificações de operações, mas nunca um conjunto de implementações de operações.Uma interface raramente aparece sozinha.

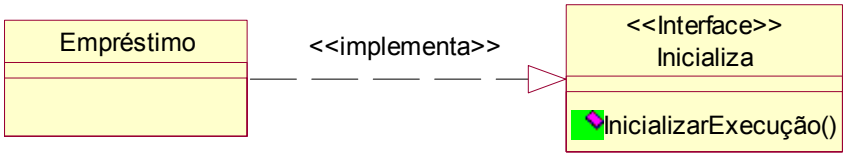


Figura 12 - Representação de Interface

4.5 Relacionamentos

Os relacionamentos interligam classes e objetos, mostrando as relações existentes entre estes. Estes relacionamentos entre as classes são mostrados no diagrama de classes. Existem basicamente três grupos de relacionamentos que são as associações, as generalizações, as dependências e os refinamentos.

4.5.1 Associações

São ligações entre classes, isto é, entre objetos e suas classes.Utilizando o conceito de UML, podemos dizer que existe uma associação quando dois objetos de tipos diferentes existem independentemente um do outro e encontram-se estreitamente ligados por um relacionamento parte-todo.As associações são usadas quando temos interesse em guardar informações de relacionamentos entre conceitos. Podemos citar um relacionamento entre usuário e livro. Precisamos saber qual instância de livro que foi emprestado está associada a uma instância de usuário que levou este livro. A partir desta associação poderemos listrar
por exemplo, quantos livros o usuário tomou emprestado, quais livros estão emprestados, qual a data do empréstimo e da devolução.

Papéis – Cada extremo de uma associação é um papel, que pode ter um nome, uma expressão de multiplicidade e uma opção de navegabilidade. A multiplicidade demonstra quantas instâncias do tipo A podem estar associadas a um tipo B. Por exemplo , a uma instância de usuário podemos ter associadas muitas instâncias de livro.

Não se deve desperdiçar muito tempo buscando associações. O mais importante na fase de análise é descobrir os conceitos do domínio do problema. Posteriormente, na fase de projeto, definiremos quais associações possuem a necessidade de implementação.

Existem diversas maneiras de implementar associações em uma linguagem orientada a objetos. A forma mais comum é usar atributos que apontam para uma instância da classe associada.

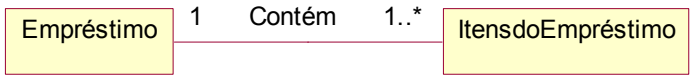


Figura 13 – Representação de Associação

4.5.2 Agregação

A agregação é uma forma especial de associação. Usando a agregação, indicamos que o relacionamento entre classes é do tipo “parte-todo”. Um computador é um exemplo de agregação, pois é formado por um conjunto de outros objetos, como teclado, mouse, monitor, cpu. Identificamos a agregação através de questionamentos do tipo “consiste de”, “contém”, “é parte de”.

Figura 14 – Representação de Agregação

4.5.3 Generalizações/Especializações

A relação deste tipo ocorre quando é possível agruparmos duas ou mais classes em uma única classe genérica. A generalização é um relacionamento entre um elemento mais geral e um mais específico. O mais específico contém informações adicionais.

Exemplo Classe Usuário e Classe Funcionário

Podemos agrupá-las em uma única classe chamada Pessoa(classe genérica), que é uma generalização da subclasses Usuário e Funcionário(classes especializadas). Como iremos utilizar a herança na implementação destas duas classes derivadas, então estas herdarão todo o comportamento e as propriedades da superclasse Pessoa. Podemos

criar uma generalização quando houver pelo menos uma propriedade que diferencie as classes que serão especializadas. No exemplo de Usuário e Funcionário, podemos dizer que o usuário terá características como Curso e o Funcionário poderá exercer uma Função, fato que o diferencia de usuário. Exemplo de generalização Pessoa/Usuário/Funcionário.

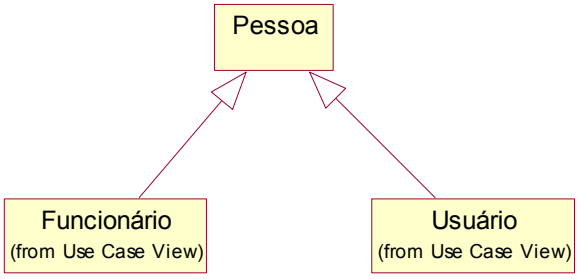


Figura 15 – Representação de Generalização

4.5.4 Composição/ Decomposição

A composição é também conhecida como agregação e permite que objetos sejam agregados para compor outros objetos ou componentes. Instâncias de objetos de uma classe são agrupadas para formar novas classes. Os componentes fazem parte do agregado. Podemos ter agregados que são compostos por outros agregados. Os componentes de um agregado poderão ou não existir fora do agregado.

A agregação é diferente da generalização porque relaciona-se a instâncias de classes de objetos, sendo chamada de “relacionamento-e” ou “relacionamento parte-de”. A generalização se relaciona a classes e é chamada “relacionamento-ou” ou “relacionamento tipo-de”. A generalização torna um objeto instância da superclasse e uma instância da subclasse. Uma universidade é uma agregação de seus vários departamentos, entretanto não é uma agregação de seus usuários pois estes são objetos independentes. Uma associação não é um fluxo de dados, nem uma conexão de objetos

O contrário, a decomposição, mostra que um objeto é composto por instâncias de outras classes. Exemplo de composição/decomposição

Figura 16 – Representação de Composição

4.5.5 Dependência

É usada para ligar dois modelos de elementos, onde um depende de outro.

É um tipo de relacionamento onde um elemento é dependente e o outro independente.

Uma mudança no elemento independente afeta o dependente. Podemos exemplificar a dependência com um pacote pk_Aluno que depende, não de forma estrutural, de uma pacote fornecedor, o pacote pk_Pessoa. A dependência pode ocorrer entre quaisquer tipos de elementos, desde que o dependente seja do mesmo tipo do independente.

4.5.6 Mecanismos Gerais

Fornecem informações extras, comentários ou semântica sobre cada elemento do modelo.

4.5.6.1 Nota

A nota é apenas um símbolo para representar restrições e comentários anexados a um elemento. Geralmente usa-se uma nota para aprimorar os diagramas. Graficamente é representada por um retângulo com um dos cantos com uma dobra na página.

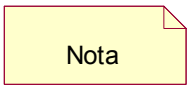


Figura 17 – Representação de Nota

4.5.6.2 Ornamentos

Servem para acrescentar um significado e são adicionados aos modelos de elementos. Podemos ”ornamentar” um elemento ao adicionarmos algo em sua representação para diferenciá-lo de outro semelhante. O nome de um tipo de elemento pode ser representado em negrito para se diferenciar de sua instância, que é escrito sublinhado. Em um relacionamento, podemos representar sua multiplicidade e desta forma identificar quantas instâncias de um tipo podem fazer parte deste relacionamento.

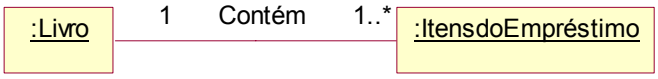


Figura 18 – Representação de um ornamento

Ao representarmos a instância da classe Livro, utilizamos :Livro(dois pontos e sublinhado) para diferenciar a instância da classe Livro.

4.6 Diagramas

A UML representa as diferentes visões do sistema através dos diagramas. Devido ao fato dos sistemas apresentarem uma estrutura estática e um comportamento dinâmico, existem diagramas específicos para representar estes modelos do sistema.

A estrutura estática do sistema é representada pelo diagrama de classes e pelo diagrama de objetos, incluindo os relacionamentos existentes entre as classes.

A estrutura dinâmica, que representa os aspectos do sistema que sofrem modificações ao longo do tempo, é descrita através dos diagramas de seqüência, de colaboração, de estado e de atividade.

É necessário representar também as transformações nos valores dos dados de um sistema, que são os aspectos funcionais do sistema. Para represenar este modelo funcional, a UML dispõe dos diagramas de componente e de execução.

Os diagramas de use case são utilizados para especificar os requisitos do sistema. Como os atores e os use cases são classes, eles também possuem associações(entre use case e ator) e generalizações.

4.6.1 Diagramas de Iteração

Criados na fase de projeto, estes diagramas são utilizados em UML para mostrar como são passadas as mensagens para que ocorra interação entre os objetos e estes cumpram suas tarefas. Podemos utilizar dois modelos de diagramas, os de Seqüência e os de Colaboração.

4.6.1.1 Diagramas de Seqüência

Os diagramas de seqüência são criados na fase de análise e auxiliam na construção do sistema desejado. Ilustram os eventos gerados pelos atores e as operações que devem ser executadas. Descrevem o comportamento do sistema, que são informações são dinâmicas, isto é, o que ele faz , não como faz,. A criação destes diagramas deve ser feita após a conclusão dos casos de uso pois dependem diretamente dos mesmos.

O diagrama de seqüência abaixo mostra uma seqüência de eventos dentro de um caso de uso, os atores externos que interagem com o sistema, o sistema e os eventos do sistema que são gerados pelos atores.

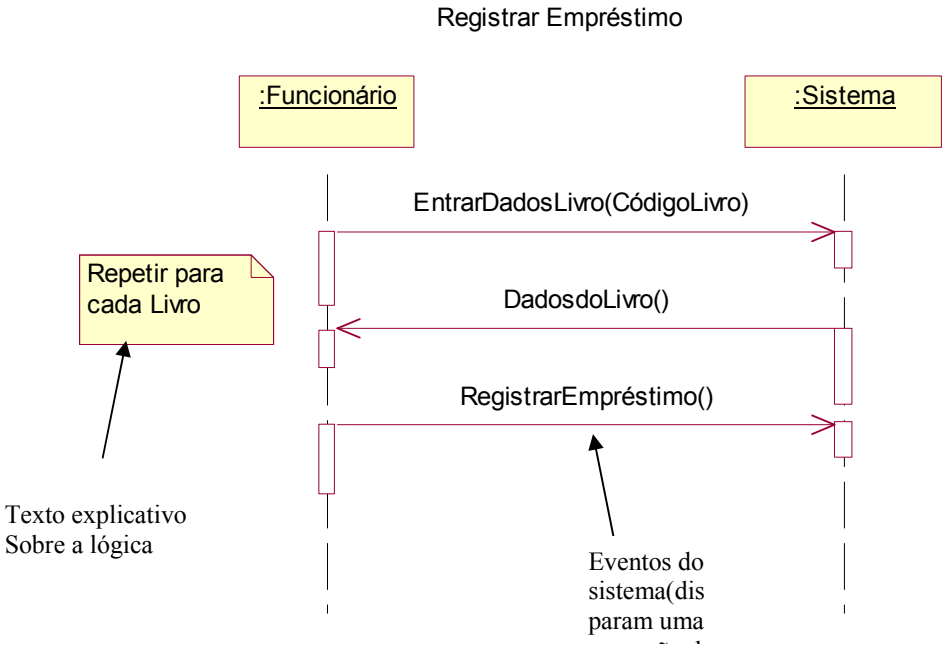


Figura 19 - Diagrama de seqüência

"O evento do sistema é um evento externo de entrada gerado por um ator para um sistema. Um evento inicia uma operação de resposta do sistema. Uma operação é a execução de uma operação do sistema em resposta a um evento do sistema."Craig Larman[1].

4.6.1.2 Diagramas de Colaboração

São descritos no formato de grafos ou redes e mostram as interações que ocorrem entre instâncias de objetos(no modelo de objetos) e entre instâncias de classes(no modelo de classes).

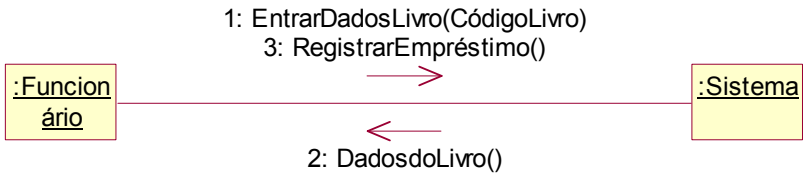


Figura 20 - Diagrama de Colaboração

4.6.2Diagrama de Casos de Uso

Os requisitos funcionais de um sistema são representados utilizando-se este tipo de diagrama. A representação é feita em função de atores externos, use-cases e o domínio do sistema modelado. Conforme dito, os atores iniciam um evento do sistema e representam o papel de uma entidade externa ao sistema, podendo ser um usuário, outro software ou um hardware. O use-case é a seqüência de ações que dispara um evento do sistema. As associações conectam atores e use-cases, que são classes, podendo se relacionarem inclusive com generalizações.

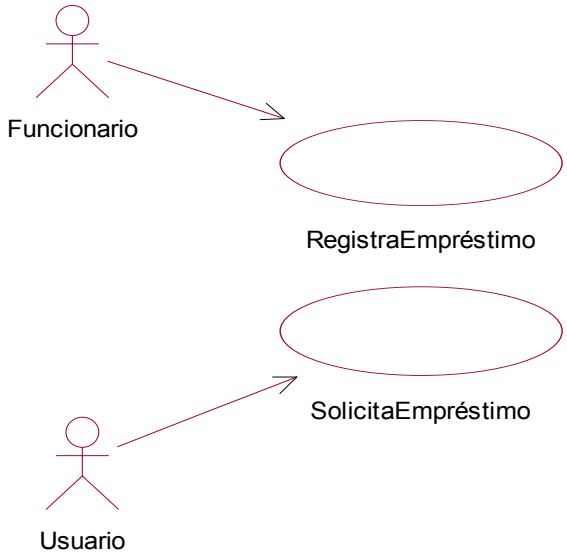


Figura 21 – Representação para um Diagrama de Casos de Uso

4.6.3 Diagrama de Classes

Demonstram especificações para as classes, suas associações, atributos e operações. Devem ser construídos após a conclusão dos diagramas de interação e do modelo conceitual, pois a partir do diagrama podemos identificar as classes necessárias e o modelo conceitual permite identificar os atributos . Devemos modelar diversos diagramas de classes para um mesmo software, podendo haver repetição de uma ou outra classe em diagramas diferentes.

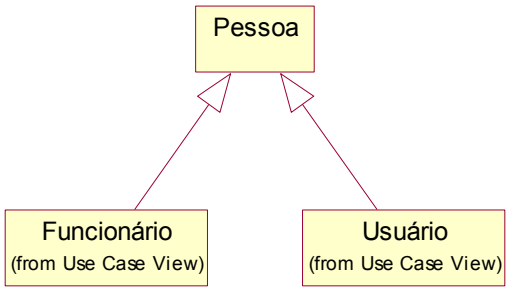


Figura 22 – Representação para Diagrama de Classes

4.6.4 Diagrama de Objetos

É uma variação do diagrama de classes, inclusive utiliza-se da mesma notação, exceto que sublinha o nome da classe para identificar que demonstra instâncias de objetos e suas associações.

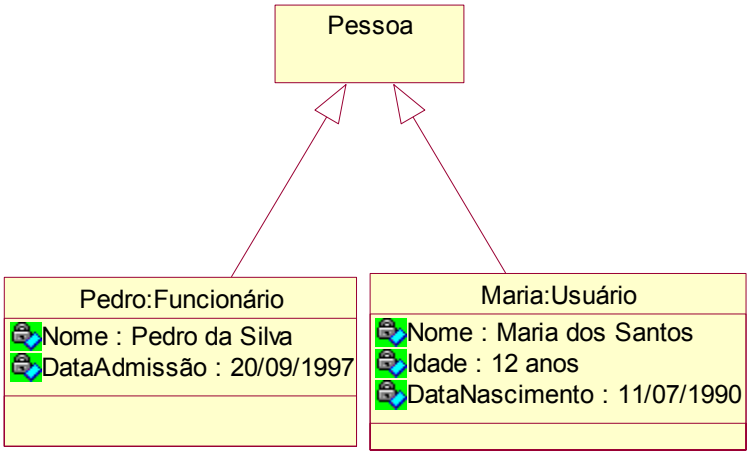


Figura 23 – Representação para um Diagrama de Objetos

4.6.5 Diagrama de Estados

O estado é a situação atual de um objeto, o valor de seus atributos e de seus relacionamentos com outros objetos. O estado é a situação do objeto dentro de um intervalo entre dois eventos recebidos por ele(o objeto). Entretanto, nem sempre todos os atributos de um objeto são alterados por um evento. A mudança de estado de um objeto devido ao novo evento é chamada de transição. O Diagrama de estados representa a sequência de eventos e o consequente estado do objeto obtido após o objeto responder a estes eventos. Este diagrama é feito para descrever o comportamento de classes de objetos do sistema, visto que , por herança todos os objetos de uma classe possuem o mesmo comportamento. Pode também ser utilizado outros elementos como tipos(conceitos) e para descrever a seqüência de eventos dos casos de uso. Devemos implementar este tipo de diagrama apenas para as classes que possuem um número conhecido e definido de estados e que possuem o comportamento afetado pelas alterações dos diferentes estados.

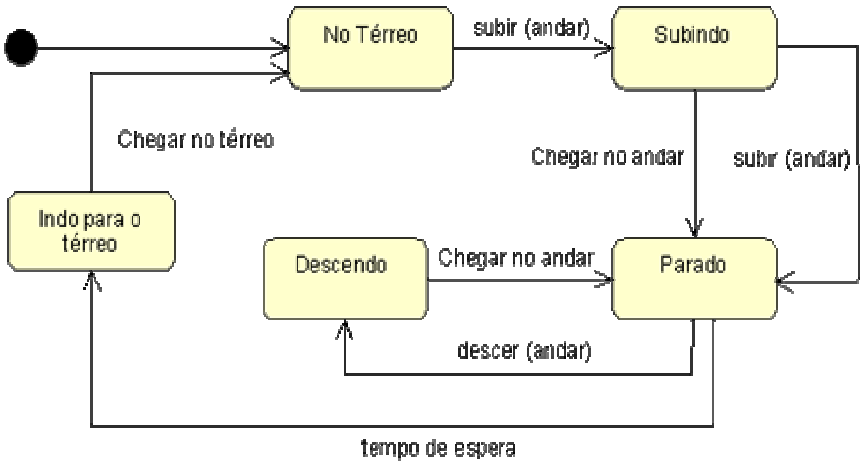


Figura 24 – Representação para um Diagrama de Estados

4.6.6 Diagrama de Atividades

É utilizado com o objetivo de demonstrar passo-a-passo, para uma operação específica do sistema, o fluxo de atividades realizadas. Representam os resultados de uma ação(método) sobre uma instância de um objeto. Este diagrama deriva do diagrama de estados, com o diferencial que é para representar as ações(atividades) para a mudança dos estados de um objeto. Podemos representar as interações, incluindo como, quando e onde as atividades são executadas por este diagrama.

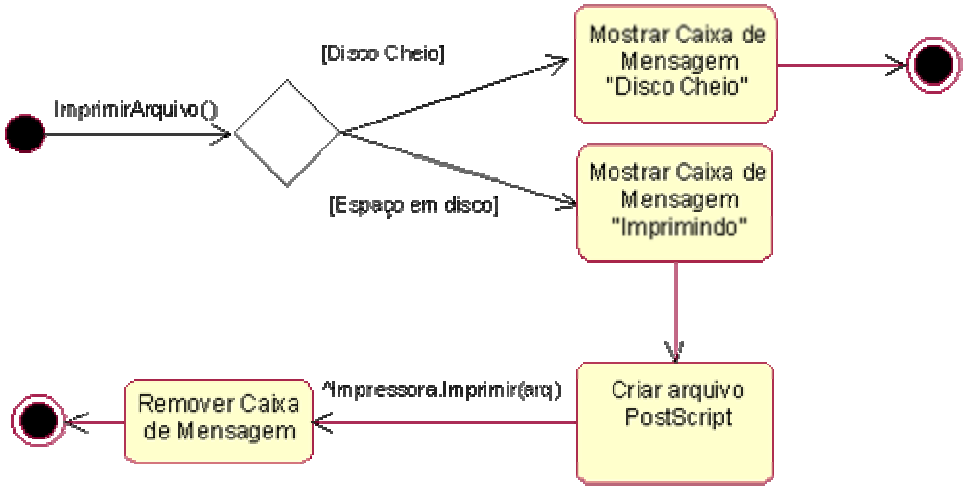


Figura 25 – Representação para um Diagrama de Atividades

4.6.7 Diagrama de Componentes

Possibilita representar os componentes de software e as dependências existentes entre eles, incluindo a representação da estrutura do código gerado. Representam os tipos definidos, como arquivos implementados em um ambiente de desenvolvimento, entretanto, estes tipos não possuem instâncias, somente componentes executam poderão tê-las. Este diagrama demonstra os conceitos lógicos, que são as classes, objetos, seus relacionamentos e suas funcionalidades implementados fisicamente(componentes) na arquitetura do sistema.

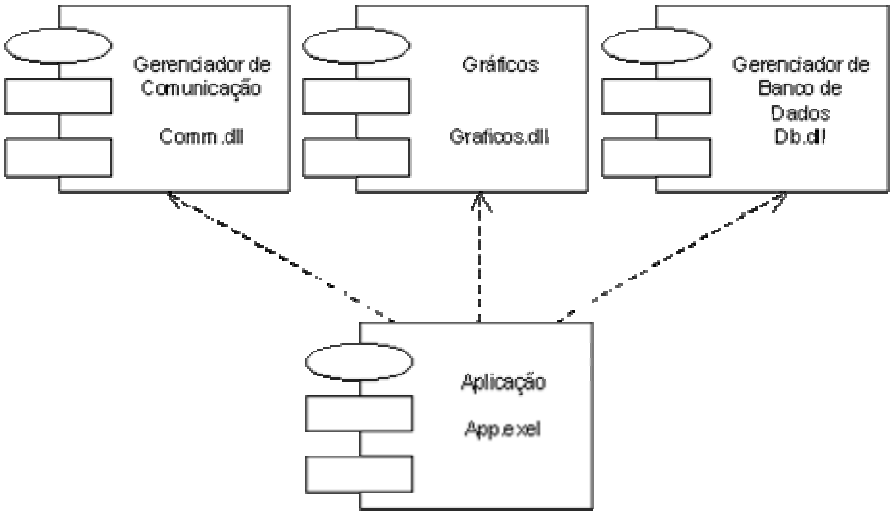


Figura 26 – Representação para um Diagrama de Componentes

4.6.8 Diagrama de Execução

O diagrama de execução demonstra a arquitetura física do hardware e do software para o sistema, incluindo-se periféricos e suas conexões. Utiliza-se o mesmo modelo de elemento, os componentes, do diagrama de componentes para representar os objetos físicos do domínio do sistema, como um host em uma LAN, um servidor e outros equipamentos para representar a arquitetura física do sistema.

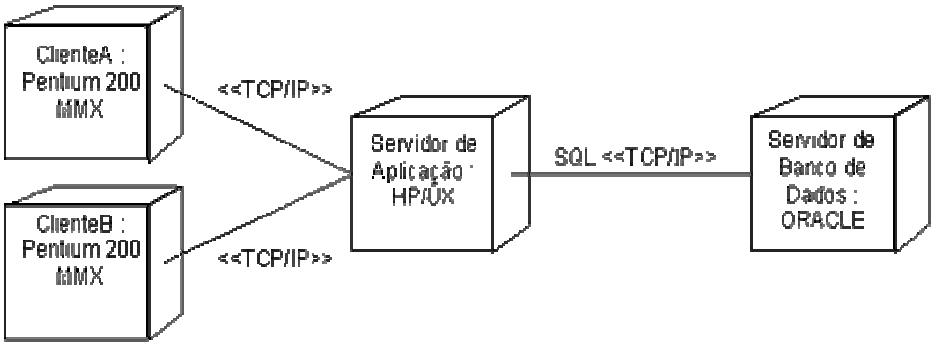


Figura 27 – Representação para um Diagrama de Execução

4.7 Ciclo de desenvolvimento

O ciclo de desenvolvimento é onde tratamos um conjunto limitado de requisitos de um ou mais casos de uso, sendo que ao término do sistema teremos passado por diversos ciclos, e em cada ciclo, por suas diversas fases.

A UML fornece um roteiro, não um método, para o desenvolvimento de sistemas, entretanto não é necessário que o roteiro seja seguido na íntegra. Podemos seguir um roteiro mínimo, que representa as funcionalidades essenciais do sistema através da UML.

O primeiro passo é a Análise de requisitos onde se deve definir claramente quais objetivos devem ser atingidos ao final da construção de um sistema.

O passo seguinte é a Análise, onde é necessário definir os mecanismos que serão utilizados para solucionar o problema. Neste ponto surgem preocupações com as abstrações usadas em UML, onde se exige a definição das classes e dos objetos.

O terceiro passo é a preocupação com o design, estendendo-se para soluções técnicas, incluindo periféricos, gerenciamento de bancos de dados e comunicação com outros sistemas. Esta fase fornece especificações para a etapa de programação.

A fase de programação é onde as classes modeladas em UML são transformadas em código. Deve-se exigir que sejam convertidas em código exatamente todas as classes que foram especificadas, para que seja refletido exatamente o que foi modelado.

A última fase é a de testes, onde o sistema é rodado em unidades do usuário.Deve-se testar a aceitação e a integração do sistema, além da verificação de que o sistema atenda os requisitos especificados.

Como exemplo, utilizamos a informatização da biblioteca da universidade. Ao definirmos qual o objetivo a ser seguido, devemos buscar o que é necessário para atingi-lo. A especificação dos requisitos é uma descrição de tudo o que é preciso para construir o produto final que irá informatizar a biblioteca.

Poderemos especificar estes requisitos da seguinte forma:

Objetivos gerais: O sistema a ser construído deverá permitir a execução de todas as tarefas necessárias em uma biblioteca no que se relaciona a empréstimo, devolução e reserva de livros.

Clientes: A biblioteca da universidade, onde o sistema será instalado em terminais.

Objetivo: Aumentar a rapidez no atendimento aos usuários da biblioteca.

Funções do Sistema: Aqui devemos descrever o que o sistema deve fazer, por exemplo, Realizar a reserva de um livro.

Atributos do Sistema: Não são funções do sistema, mas características, como rapidez no tempo de resposta quando uma consulta é realizada, facilidade de uso, tolerância a falhas.

4.8 Fases de Desenvolvimento de Sistemas

Basicamente, podemos dividir as fases de desenvolvimento de sistemas em cinco:

Análise de requisitos, Análise, Design, Programação e Testes.

4.8.1 Análise de Requisitos

O passo número um e o mais importante ao iniciarmos a construção de um sistema é a Análise de Requisitos. É onde definimos os casos de uso que irão retratar os requerimentos do cliente. O caso de uso descreve uma funcionalidade fornecida pelo sistema. Dessa forma, devemos implementar nesta etapa os casos de usos e seus relacionamentos com os atores, que serão representados em diagramas de casos de uso. A preocupação é apenas em demonstrar os requerimentos do cliente, “o que” ele deseja, sem considerações no que diz respeito a “como” implementar. Os diagramas de seqüência e de estado do sistema também são implementados nesta fase.

4.8.2 Análise

A fase seguinte é a de análise, onde surgem preocupações com representação das abstrações da UML, como classes, objetos, os relacionamentos e colaborações entre

estas classes. Nesta fase iremos construir o diagrama de Classes e iniciar a construção do modelo conceitual. Devemos modelar apenas as classes que irão fazer parte do domínio do problema e não classes técnicas com definições de detalhes e soluções do software, como classes de interface com o usuário e de definições de bancos de dados ou de comunicação.

A fase analisar é iniciada após a identificação e documentação dos casos de uso. Nesta fase, iremos investigar os problemas do ciclo corrente. Devemos sempre lembrar em mostrar apenas soluções para o domínio do problema. Para obter este resultado, inicialmente devemos projetar o modelo conceitual, que irá identificar os conceitos no domínio do sistema. Podemos implementar o modelo conceitual em partes, acrescentando novos conceitos a cada ciclo, à medida que formos analisando cada caso de uso. Podemos também realizar refinamentos de casos de uso quando houver necessidade

4.8.2.1 Modelo Conceitual

Durante a fase de análise, procura-se decompor o problema em conceitos e objetos, por este motivo, o modelo conceitual é considerado o artefato mais importante a ser criado nesta fase , pois é este modelo que demonstra os conceitos em um domínio do problema. Em UML utilizamos um diagrama estático para representar o modelo conceitual, pois nele iremos representar conceitos do mundo real. Durante o desenvolvimento deste diagrama, é ideal que estejam prontos os casos de uso e documentos auxiliares para que seja possível identificar os conceitos que serão representados no modelo. O modelo conceitual não representa nenhuma operação ou modelo de software, apenas conceitos, associações entre estes conceitos e atributos dos conceitos.

Um conceito é uma idéia, uma coisa ou um objeto que é representado por um símbolo e é definido por uma intenção. A extensão é um conjunto de exemplos aos quais podemos aplicar este conceito. Como exemplo, podemos representar o acervo de livros da biblioteca por um símbolo **Livros**. A intenção do conceito Livros é representar o conjunto de todos os livros da biblioteca. Devemos documentar todos os conceitos que acharmos necessário e até mesmo aqueles que se acredita dispensáveis. A identificação dos conceitos deve ser feita buscando-se substantivos e frases nos casos de uso e em

documentos, que podem representar conceitos ou atributos para inclusão no modelo conceitual.

Podemos utilizar a seguinte seqüência para construir um modelo conceitual

- Listar os conceitos candidatos;
- Incluir estes conceitos no modelo conceitual;
- Incluir associações para representar os relacionamentos que se deseja guardar informações;
- Adicionar os atributos para representar os requisitos de informações.

Para distinguir atributos de conceitos devemos pensar da seguinte forma, Conforme descreve Larman[1] :

"Se você pensa em um conceito X como um número ou um texto no mundo real, então X provavelmente é um conceito e não um atributo."

Construção do Modelo Conceitual:

- Inicialmente, devemos identificar os conceitos necessários para atender os requisitos do sistema. Para buscar estes conceitos, utilizaremos os casos de uso já definidos que são registrar Devolução no prazo e Registrar Empréstimo. Nestes casos de uso buscaremos objetos, descrições de coisas, lugares, transações, papéis desempenhados por pessoas que são relevantes para acrescentar ao modelo.

Caso de Uso : Registrar Devolução no prazo

Ações do Ator	Resposta do Sistema
1. O **funcionário** recebe do **usuário** o(s) **livro(s)** que levou emprestado.	
2. O funcionário registra no sistema os dados do livro e do usuário para verificar se a **devolução** está no prazo.	**3.** O sistema emite dados identificando que a devolução está dentro do prazo.
4. O funcionário confirma a devolução.	**5.** O sistema emite comprovante de devolução com os dados da devolução.

Revisando o caso de uso acima, podemos identificar os seguintes conceitos: funcionário(realiza um papel), usuário (realiza um papel),livro(objeto),devolução(é uma transação).

Caso de Uso : Registrar Empréstimo

Ações do Ator	**Resposta do Sistema**
1. O **funcionário** recebe do usuário o(s) **livro**(s) que deseja levar emprestado.	
2. O funcionário registra no sistema os dados do livro para verificar se é possível realizar o **empréstimo**	**3.** O sistema emite dados identificando que o livro é para empréstimo (ou para consulta).
4. Considerando que o livro é para empréstimo, o funcionário registra os dados do usuário.	**5.** O sistema emite confirmação de que o usuário está apto a obter o empréstimo.
6. O funcionário confirma o empréstimo.	**7.** O sistema registra os dados e emite comprovante de empréstimo para o usuário com dados do usuário, dados do livro e a data do empréstimo .

O caso de uso acima acrescenta o conceito de empréstimo, que é uma transação.
Podemos dizer que os conceitos significativos do domínio da biblioteca são basicamente Livro, Funcionário, Usuário e Empréstimo.

Modelo Conceitual

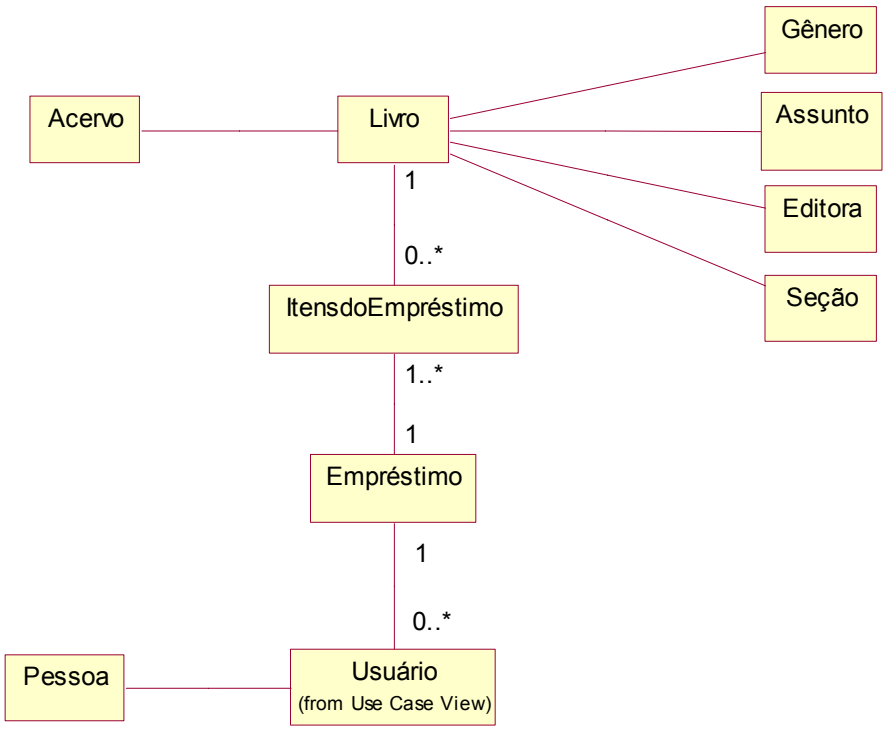


Figura 28 – Representação de Modelo Conceitual

4.8.3 Design

A terceira fase é o Design, onde os dados obtidos na Análise serão repassados para implementações técnicas. Esta etapa é equivalente à fase projetar, conforme descrita por Larmam[1].Podemos incluir interfaces, especificações de periféricos, soluções para gerenciamento de bancos de dados e para a comunicação com outros sistemas, enfim, tudo o que está relacionado às especificações de infra-estrutura para a fase de programação. Deve-se construir nesta fase os diagramas de pacotes, refinar os diagramas de interação, de casos de uso, de classes e de estados.

4.8.4 Programação

A fase seguinte, de Programação ou construção, será iniciada somente após finalizadas as definições das classes em UML. Nesta etapa as classes serão convertidas para o código fonte da linguagem orientada a objetos que foi escolhida. É extremamente necessário que seja implementado para código fonte o que foi realmente definido na fase anterior, pois isto faz com que o sistema esteja de acordo com as especificações. Ao dar seqüência ao andamento desta fase, podemos utilizar os padrões sugeridos por Larman[1] para obter um software bem projetado.

4.8.5 Testes

A fase final do desenvolvimento é a de Testes. O objetivo desta etapa é verificar se o sistema possui a funcionalidade especificada pelo consumidor.Neste momento devem ser realizados testes de integração, pelo programador e testes de aceitação pelo cliente nas unidades onde o sistema será implantado. Os testes devem ser feitos em blocos separados, testando inicialmente as classes e fazendo uso de diagramas e das definições das classes . Logo em seguida, deve-se integrar estes testes e testar o sistema como um todo, uma “caixa preta”, para verificar se ele realmente está realizando o que foi proposto.
O teste de integração utilizam, em geral, diagramas de componentes e diagramas e colaboração. Os testes de sistemas baseiam-se nos diagramas de caso de uso para realizar a validação do sistema.
Estas etapas de desenvolvimento deverão ser realizadas dentro de um ciclo de desenvolvimento, passando-se por diversos ciclos até a finalização do software.

Padrões

Os padrões surgiram da observação de que, durante o desenvolvimento de sistemas, existe a repetição de certos tipos de problemas. O interessante é que estes problemas, por serem bastante parecidos, podem ser documentados em pares problema/solução. A solução é a descrição da forma geral que seria aplicada para resolvê-los. A solução do problema é obtida através da reutilização a longo prazo de boas soluções.
A vantagem de aplicarmos um padrão para resolver um problema é evitar o desperdício de tempo na busca da melhor solução , pois é justamente isso que o padrão disponibiliza. A clareza das informacões, a facilidade para documentação e para o aprendizado por desenvolvedores novatos também é outro bom motivo para a utilização de padrões de desenvolvimento.
Atualmente, a intenção dos padrões é, além de propor a melhor solução para um problema e sua reutilização, compartilhar conhecimento entre desenvolvedores de sistemas para documentar pares de problemas e suas respectivas soluções, obtendo-se uma linguagem comum.

A idéia de padrão surgiu através de trabalhos de Christopher Alexander, que era arquiteto e verificou a necessidade de uma padronização nos projetos com o objetivo de melhorar a qualidade de vida das pessoas.

O surgimento dos padrões, ou Design Patterns, para desenvolvimento de sistemas ocorreu no final dos anos 80. Os primeiros padrões e suas aplicações foram desenvolvidas por Ward Cunningham e Kent Beck para desenvolvimento de interfaces em Smalltalk. Na mesma época, Jim Coplien desenvolvia um catálogo de padrões em C++ e Erich Gamma procurava documentar estruturas de projetos que apareciam repetidamente durante o desenvolvimento. O primeiro catálogo de padrões surgiu em 1993.

Um design pattern é composto de um par problema/solução em um determinado contexto. O problema é o que se deseja resolver dentro de um contexto. A solução é a proposta de como resolver este problema aplicando-se a melhor solução encontrada por desenvolvedores experientes. O contexto é o conjunto onde se aplica o par problema/solução. No domínio da orientação a objetos, um padrão identifica classes e

objetos que se comunicam e que são utilizadas para resolver problemas repetitivos dentro de um contexto.

Um padrão não significa necessariamente nenhuma novidade para o desenvolvimento de um sistema. Em geral, um padrão demonstra uma prática comum, amplamente testada e que funciona. Oferece uma técnica, com princípios de desenvolvimento que são seguidos como uma rotina, uma regra. Conforme Larman[1], um padrão "não propõe nenhuma nova idéia, trata-se exatamente do oposto - eles são uma codificação de princípios básicos existentes e amplamente utilizados". Um padrão é representado através de pares problema/solução, onde o problema/solução é descrito de forma genérica para que se possa extrair os possíveis casos onde poderemos aplicá-los.

5.1 Grasp

Os padrões Grasp, desenvolvidos por Craig Larman, têm como objetivo principal definir as responsabilidades de um objeto do sistema.

Segundo Larman[1], "as responsabilidades estão relacionadas às obrigações de um objeto em termos do seu comportamento". As responsabilidades principais são "conhecer" e "fazer". O objeto tem a responsabilidade de fazer alguma coisa por ele mesmo, dar início em ações em diferentes objetos e também controlar atividades de outros objetos. A responsabilidade de conhecer implica em obter dados privados de outras classes, conhecer objetos relacionados a ele e derivar ou calcular dados de outros objetos. No sistema de biblioteca, podemos definir que um empréstimo tem a responsabilidade de "fazer" a impressão dos dados do empréstimo. Um método corresponde à implementação de responsabilidade. Ao imprimir os dados do empréstimo, o método ImprimirEmpréstimo() irá satisfazer a responsabilidade da classe Empréstimo de imprimir seus próprios dados. Podemos utilizar os diagramas de interação para demonstrar a atribuição de responsabilidades a objetos.

5.1.1 Expert

O padrão Expert ou especialista apresenta o princípio de que a classe especialista, isto é, a que tem o domínio da informação é quem deve ter a responsabilidade de trabalhar com estes dados. Por exemplo, devemos atribuir a responsabilidade de imprimir os dados do empréstimo à classe Empréstimo, porque esta classe possui todos os dados necessários. Podemos investigar qual a classe é a expert nos dados do empréstimo utilizando perguntas como “Quem conhece os dados do empréstimo?” A resposta é simples, a própria classe empréstimo, pois ela é especialista na informação.

Há casos em que apenas uma classe não poderá ser a responsável por toda a tarefa e necessitará de outras classes, que conhecem parcialmente alguns dados.

Podemos observar que para imprimir os dados do empréstimo também são necessários dados do livro, como Código e Título. A classe Livro é a especialista nestas informações, portanto, é ela quem deverá fornecer estes dados através de uma mensagem enviada a ela pela classe empréstimo. Neste caso, dizemos que a classe livro é uma especialista “parcial” .

A vantagem da utilização deste padrão é que, ao deixar a classe expert responsável pela obtenção dos dados, estamos encapsulando a informação na classe que a possui.

5.1.2 Creator

Este padrão tem por princípio atribuir a responsabilidade de criação de uma nova instância de um objeto A a uma classe que contém,agrega ou usa objetos desta classe. Esta tarefa é muito comum em sistemas OO, daí a necessidade de um padrão que definisse quem deve criar uma instância de um objeto. Em geral, relacionamentos de agregação demonstram as classes que agregam, contêm ou usam um objeto.

O padrão expert poderá servir de base para definirmos a responsabilidade da criação, pois pode-se atribuir esta responsabilidade à classe que possui os dados de inicialização necessários à criação do novo objeto. Os dados iniciais são passados durante a criação através de um método de inicialização. Em delphi e em Java utilizamos um construtor que pode ou não receber parâmetros.

Podemos questionar quem deve ser o responsável pela criação de uma nova instância da classe ItensdoEmpréstimo. A classe Empréstimo é mais indicada por ser a conhecedora de dados iniciais dos itens pertencentes ao empréstimo em questão.

Os padrões apresentados, expert e criator(criador) relacionam-se com outros padrões que serão vistos a seguir.

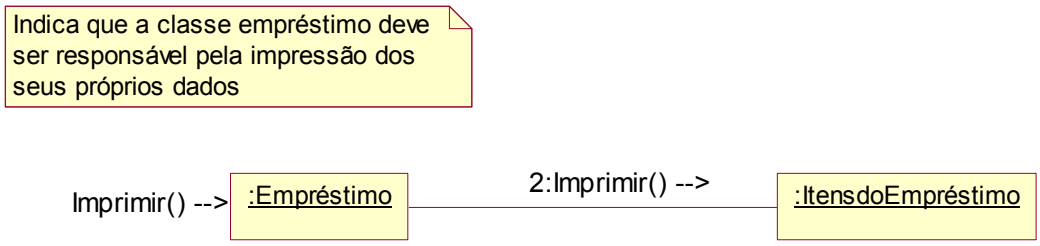


Figura 29 – Representação da Atribuição de responsabilidade

5.1.3 Low Coupling

O acoplamento fraco busca resolver o problema da reutilização de uma classe. O acoplamento de uma classe refere-se ao quanto esta classe depende de outra. Quanto menos uma classe "depende" de outra, mais fracamente acoplada ela é. Significa dizer que, quanto mais independente for a implementação de uma classe A, mais fácil é sua manutenção pois uma mudança nas classes relacionadas não implica muitas mudanças em A. Também podemos dizer que mais fácil é a sua reutilização por outras classes, pois sua implementação a torna independente da implementação das outras classes.

Como exemplo a criação de uma nova instância de Assunto para uso da classe Livro conforme o modelo conceitual, poderia ser atribuída à classe Acervo. Neste caso, seria preciso que acervo conhecesse a classe Assunto, o que a tornaria acoplada à Assunto. O ideal é que a classe Livro seja a responsável pela criação de nova instância de Assunto, pois Livro conhece Assunto. O mesmo raciocínio irá servir para Gênero e Editora. Desta forma estaremos favorecendo o baixo acoplamento da classe Acervo e tornando-a mais independente, sendo necessário apenas que ela conheça a classe Livro . Este padrão não é útil separadamente. É sempre utilizado em conjunto com outros, como o creator e o expert e é interessante em projetos que buscam a reutilização de componentes.

5.1.4 High Cohesion

A coesão mede quanto as responsabilidades de uma classe estão relacionadas. É desejável que as classes possuam alta coesão, ou seja, possuam responsabilidades altamentes relacionadas, mas que não executem uma enorme diversidade de funções.

O objetivo deste padrão, também chamado de Alta Coesão, é atribuir a responsabilidade de execução de uma tarefa a um objeto mantendo esta alta coesão. Ao diversificarmos demais as responsabilidades de uma classe, sem que estas atividades estejam relacionadas, tornamos a classe suscetível às mudanças, dificultando sua manutenção e sua reutilização. O interessante é que uma classe deva executar somente o que está ao seu alcance, aquilo que ela tem conhecimento para executar. Ao atribuirmos a responsabilidade de Livro criar novo Assunto, favorecemos a alta coesão desta classe(Livro), pois Livro possui os dados necessários e iniciais para a nova instância e está diretamente relacionada à classe Assunto.

5.1.5 Controller

O padrão controller objetiva atribuir a responsabilidade de tratamento de eventos do sistema a uma classe representativa de todo o sistema. É um “controlador fachada”, que tem a responsabilidade de tratar todos os eventos externos de um ator, provenientes de um caso de uso. A classe escolhida para ser o controller deve representar todo o sistema,

Se existirem poucos eventos , pode representar o negócio ou é simplesmente um tratador dos eventos de um caso de uso. A opção de implementar um controller para cada caso de uso é ideal, pois se novas opções de casos de uso forem acrescentadas, não teremos uma única classe controladora grande e com pouca coesão.

A utilização de um controller para tratamento de eventos do sistema aumenta a possibilidade de reutilização das outras classes, pois ficam apenas com a responsabilidade de executar tarefas dentro do seu domínio. O controller deve ser implementado de forma a delegar as tarefas a outros objetos do sistema e utilizando-se também dos conceitos de outros padrões, como os descritos anteriormente.

Para a implementação da biblioteca, poderíamos implementar uma classe controller chamada “Principal” para tratar os eventos do funcionário, que é um ator externo ao sistema e que, por exemplo ao indicar para o sistema finalizar um empréstimo pressionando um botão, gera um evento que faz com que o empréstimo seja efetuado e

imprima os seus dados. Observe que o controller não irá executar as operações, mas apenas tratar os eventos e enviá-los às classes que são experts na realização da tarefa.
É importante salientar que a interface, ou camada de apresentação também não deve tratar eventos do sistema, mas encaminhar a tarefa ao controller e este à classe especializada na informação.

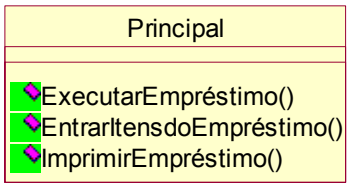


Figura 30 – Representação de uma classe para Controller.

Conclusão

A OO em conjunto com a UML serão empregadas constantemente na análise e implementação de sistemas. Devido ao fato da UML ser uma técnica bastante flexível, o que permite aperfeiçoamentos nos seus conceitos, ela já é empregada no desenvolvimento de ferramentas de modelagem visual e em outros ambientes de desenvolvimento, o que faz com que esta técnica seja utilizada como linguagem de modelagem padrão nos mais diversos tipos de sistemas e ferramentas para desenvolvimento. A UML, os padrões de desenvolvimento e os frameworks que se utilizam da OO são a grande solução para diminuir o tempo de desenvolvimento de qualquer sistema, para qualquer área do conhecimento humano.
Os padrões,a UML e a OO são indispensáveis, mas somente a utilização destas técnicas não nos permite obter o menor tempo na criação de soluções informatizadas. É intessante melhorar o processo de desenvolvimento de software de forma a torná-lo completamente padronizado, podendo-se para isso utilizar as idéias introduzidas pelo CMM, que possibilita agregar mais qualidade ao software.
Atualmente grandes empresas se mobilizam para criar frameworks utilizando UML, OO e padrões, como o Template Method, porque suas classes são altamente reutilizáveis.

Referências Bibliográficas

[1] Larman, Craig.''**Utilizando UML e Padrões : uma introdução à análise e ao projeto orientados a objetos**''. trad. Luiz A. Meirelles Salgado. Porto Alegre: Bookman, 2000.

[2] Rumbaugh, James; Blaha, Michael; Premerlani, Willian; Eddy,Frederick; Willian Lorensen."Modelagem e Projetos Baseados em Objetos". - tradução de Dalton Conde de Alencar. Rio de Janeiro: Campus,1994.

[3] Fowler, Martin.''**UML Destiled : A brief guide to the standard object modeling language**''**.** 2ª ed. Addison Wesley, 2000.

[4] Barros, Pablo Fernando do Rêgo. ''**Monografia : Uml Linguagem de Modelagem Unificada**''**.**

[5] Gamma, Erich; Helm, Richard; Johnson,Ralph; Vlissides, John. ''**Padrões de Projeto: Soluções Reutilizáveis de Software Orientado a Objetos**''**.** Porto Alegre: Bookman, 2000.

[6] Medeiros, Álvaro. ''**Visão Geral de UML**''**.**

Artigos e sites consultados na Internet

Orientação a Objetos http:// www.tool.pro.com.br

Rational http://www.rational.com

Rumbaugh

http://www.sigs.com/publications/docs/9601/oc9601.c.chonoles.html#OMT

Object FAQ http://iamwww.unibe.ch/~seg/OOinfo/FAQ/index.html

Booch
http://www.sigs.com/publications/docs/9601/oc9601.c.chonoles.