Projeto de Software

UNIDADE 1

Orientação a Objetos e Modelos de Processo Iterativos e Incrementais: O Processo Unificado

Autores:

Profa. Dra Rosana Terezinha Vaccare Braga
Prof. Dr. Paulo Cesar Masiero

ICMC/USP

Professor da Disciplina:
Prof. Dr. Valter Camargo
Prof. Esp. Alexandre Pedroso Fernandes

Projeto de Software

1 Introdução

1.1 – Contexto e motivação para modelagem orientada a objetos:

O que é análise Orientada a Objetos?

Para responder a esta pergunta, convém discutir um pouco o processo de desenvolvimento de sistemas. Em geral, antes que um sistema possa ser desenvolvido, é necessário saber seus requisitos e passar por uma fase de análise, com o objetivo de criar modelos que representem o sistema em formatos mais facilmente implementáveis. Esse esforço é necessário e vale a pena, pois, para garantir a qualidade do sistema resultante é necessário seguir um processo de desenvolvimento de software.

Iniciar a programação diretamente, **sem passar pela fase de análise**, como se faz frequentemente quando se desenvolve um sistema pequeno e simples, pode ser desastroso quando o sistema alvo é grande e/ou complexo. Pode-se ocasionalmente obter um sistema satisfatório e que consiga sobreviver ao desafio da manutenção, mas isso certamente seria um caso isolado e dependente da sorte e de competências individuais.

Quando se segue um processo de desenvolvimento, seja ele qual for, uma fase importante refere-se à análise do sistema. Essa fase é tratada em uma parte do Processo Unificado denominada "**Disciplina de Requisitos**". O Processo Unificado é o processo que será usado neste curso. Os modelos produzidos durante essa disciplina são depois transformados em modelos de projeto e, finalmente, em código fonte. Os modelos de análise são mais próximos do que ocorre no mundo real, enquanto os modelos de projeto vão um pouco além e refletem mais concretamente as soluções computacionais para tornar a implementação possível.

A **análise orientada a objetos** oferece diversas opções de modelos para que as informações sobre o sistema a ser desenvolvido sejam representadas de forma mais semelhante possível aos objetos presentes no mundo real. Assim, a análise e projeto de software orientados a objetos conduzem a modelos contendo vários objetos, muitos dos quais correspondem diretamente a elementos do sistema do mundo real. Por exemplo, para representar uma biblioteca por meio de um sistema orientado a objetos, o modelo teria objetos como Livro, Leitor, Atendente, Bibliotecária, etc.

A **análise orientada a objetos** fornece mecanismos para representação de modelos estáticos e dinâmicos. Modelos estáticos podem mostrar, por exemplo, os objetos do sistema e seus relacionamentos. Já os modelos dinâmicos mostram a comunicação entre os objetos para alcançar os propósitos do sistema. Neste caso, é importante identificar a ordem em que as atividades são desempenhadas para cumprir certo objetivo.

Por exemplo, um modelo estático de uma biblioteca mostraria o relacionamento entre seus diversos componentes. Biblioteca possui Estantes que armazenam Livros, que são emprestados pelo Atendente ao Leitor. Um modelo dinâmico poderia mostrar o processo de empréstimo de um livro a um leitor. O Leitor dirige-se ao balcão e entrega ao atendente o livro a ser emprestado. O Atendente solicita os dados do leitor, calcula a data de devolução e registra o empréstimo.

1.2 – Histórico da Orientação a objetos

A Orientação a Objetos (OO) é um paradigma para a produção de modelos que especifiquem o domínio do problema de um sistema. Quando construídos corretamente, sistemas orientados a objetos são flexíveis a mudanças, possuem estruturas bem conhecidas e oferecem a oportunidade de criar e implementar componentes reutilizáveis. Componentes reutilizáveis são componentes que já foram **testados e aprovados** e que podem ser reutilizados durante o desenvolvimento de novas aplicações. Modelos orientados a objetos são implementados convenientemente utilizando uma linguagem de programação orientada a objetos.

A OO surgiu no fim da década de 60, quando dois cientistas noruegueses, Ole-Johan Dahl e Kristen Nygaard, criaram a linguagem Simula (Simulation Language). Simula I (1962-65) e Simula 67 (1967), as primeiras linguagens OO, eram usadas para simulações do comportamento de partículas de gases. Os conceitos de objetos, classes e herança já estavam presentes, embora não necessariamente com estes nomes. A herança apareceu apenas em Simula 67 e se falava apenas em criação de subclasses. Simula passou a ser usada com freqüência na década de 70, sendo inclusive a linguagem utilizada para desenvolver as primeiras versões de Smalltalk, logo no início da década de 70, quando surgiu o termo "**Programação Orientada a Objeto**" (POO).

As principais características de OO, descritas mais adiante, como por exemplo, classes, objetos, encapsulamento, herança e polimorfismo, passaram a ser incorporadas nas linguagens de programação existentes, como por exemplo em Ada, BASIC, Lisp e Pascal. Mas adicionar características OO a linguagens que não haviam sido projetadas para esse fim levou à problemas de compatibilidade e manutenibilidade de código. Por outro lado, linguagens de programação puramente OO não possuíam certas características que os programadores passaram a exigir. Isso levou à criação de novas linguagens baseadas em métodos OO, mas permitindo algumas características procedimentais em locais seguros. Eiffel é um exemplo de linguagem relativamente bem sucedida criada com esses objetivos. Mais recentemente surgiram várias outras linguagens desse tipo, como por exemplo, Python e Ruby. Além de Java, algumas das linguagens orientadas a objetos atuais com maior importância comercial são VB.NET e C Sharp (C#), projetada para a plataforma .NET da Microsoft.

Na metade da década de 80, quando as linguagens orientadas a objetos começaram a fazer sucesso, com grande influência de C++, que é uma extensão de C, surgiu a necessidade de processos para dar suporte ao desenvolvimento de software orientado a objetos. Na verdade, os

processos de desenvolvimento utilizados na época eram os relacionados ao paradigma da análise estruturada. Realizar a análise estruturada de um sistema e depois programá-lo usando uma linguagem orientada a objetos era muito trabalhoso, pois os modelos resultantes da análise estruturada precisavam de muitas adaptações até que pudessem ser implementados usando OO. Assim, o surgimento da orientação a objetos exigiu a criação de processos que integrassem o processo de desenvolvimento e a linguagem de modelagem, por meio de técnicas e ferramentas adequadas.

Abordagens como OMT [Rumbaugh, 1990], Booch [Booch, 1995] e Fusion [Coleman et al, 1994] foram as primeiras a surgir para dar suporte à análise e projeto OO. Elas estabelecem processos por meio dos quais é possível partir de um problema do mundo real, obter diversos modelos de análise, derivar os modelos de projeto e chegar às classes a serem implementadas.

Depois de quase uma década do surgimento das linguagens OO, estabeleceu-se uma gama enorme de processos de desenvolvimento OO, o que passou a dificultar a comunicação entre analistas e projetistas de software. Em geral, cada processo propunha sua própria notação para OO, com símbolos diferentes para denotar herança, agregação, associações etc. A UML (Unified Modeling Language) [Rational, 2000] – Linguagem de Modelagem Unificada – surgiu com o intuito de criar uma notação **completa e padronizada**, que todos pudessem usar para **documentar** o desenvolvimento de software OO.

A Figura 1 mostra os modelos de cada um dos processos de análise existentes anteriormente e que serviram como base para a criação da UML (em itálico mostram-se os nomes dos diagramas correspondentes na UML).

No entanto, a UML não apresenta um processo, mas apenas a notação. Por isso, alguns anos depois de sua criação, passaram a surgir propostas de processos de desenvolvimento com base na UML. Exemplos são o PU (Processo Unificado) [Larman, 2004] e sua especialização pela Rational, o RUP (Rational Unified Process) [Krutchen, 2000]. Neste curso adotaremos o PU como sendo o processo pelo qual partimos dos requisitos de um software e chegamos projeto.

1.3 – Conceitos básicos do Paradigma Orientado a Objetos

Alguns conceitos básicos sobre o paradigma OO são introduzidos nesta seção e serão refinados ao longo do texto, na medida em que forem necessários. São conceitos considerados fundamentais para o entendimento da abordagem OO, tais como classes, objetos, abstração, encapsulamento e herança.

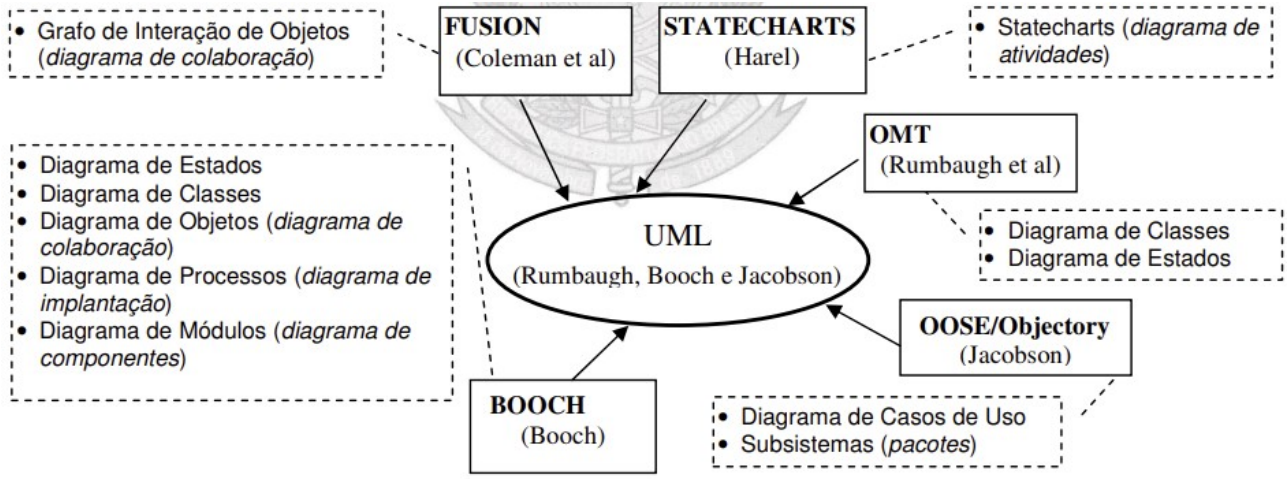

Figura 1 – Derivação da UML.

1.3.1 – Classes e Objetos

O princípio mais básico do paradigma OO é a organização de **Objetos** do sistema em **Classes**. **Objetos** podem ser definidos como entidades individuais que tem características e comportamento em comum. Por exemplo, o "Leitor" de uma biblioteca possui atributos tais como nome, RA (Registro do Aluno), data de nascimento, curso etc. Possui também comportamento, como "registrar-se na biblioteca", "emprestar livro", "devolver livro" etc. Quando se deseja falar de um conjunto de objetos semelhantes sem ter que falar de cada um individualmente, fala-se da **Classe** a que esses objetos pertencem. Por exemplo, podemos falar da classe "Leitor", que contém todos os leitores da biblioteca da UFSCar. Cada leitor em particular, como por exemplo João da Silva, é dito ser um objeto da classe 'Leitor" (ver Figura 2). Um outro exemplo: para representar a cozinha de uma casa por meio de um sistema orientado a objetos, o modelo teria objetos como Mesa, Cadeira, Geladeira, Fogão, Forno de Microondas, Armário, Balcão, Cozinheiro, Pessoa, Despensa, Utensílios, Alimento, etc. O Forno de Microondas possui atributos como capacidade, potência, status, hora, etc. Possui também comportamento, como ligar, programar, mudar potência, descongelar, desligar, etc. Em OO, o comportamento é implementado por meio de métodos de cada classe, o que será tratado mais adiante neste curso.

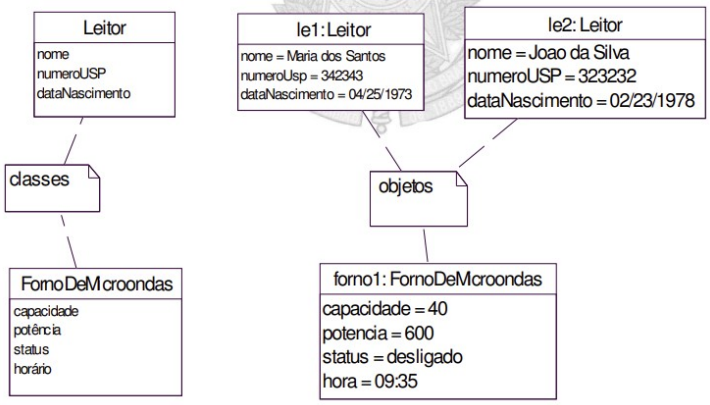

Figura 2 – Classes X Objetos.

1.3.2 – Abstração

Abstração é o processo pelo qual conceitos gerais são formulados a partir de conceitos específicos. Em outras palavras, um conceito geral é formado pela e**xtração de características comuns de exemplos específicos**. Assim, em uma abstração, detalhes são ignorados, para nos concentrarmos nas características essenciais dos objetos de uma coleção. O conceito de abstração é vital no paradigma OO, pois permite a **criação de classes por meio das características e comportamentos comuns a diversos objetos**.

A abstração utiliza uma estratégia de **simplificação** de detalhes. Como detalhes concretos podem permanecer ambíguos, vagos ou indefinidos, para que a comunicação em um nível abstrato tenha sucesso é necessário que haja uma experiência comum ou intuitiva entre os interlocutores. Por exemplo, é diferente uma conversa entre os arquitetos que irão projetar uma cozinha, preocupados com dimensões, usabilidade da cozinha etc., e entre o cozinheiro e o proprietário da casa, que estão mais preocupados com as receitas a serem elaboradas durante a semana.

A abstração nos leva a representar os objetos de acordo com o ponto de vista e interesse de quem os representa. Por exemplo, no caso de nossa cozinha, descrevemos um Forno por meio de sua capacidade, potência, status, hora, etc. Essas são as características que nos interessam para que o forno seja considerado útil na preparação de alimentos. **Ao fazermos a identificação de um forno a partir dessas características, estamos fazendo uma abstração, já que não estamos considerando muitos outros detalhes necessários para descrever totalmente um Forno**. Para um vendedor de Fornos, outras características seriam necessárias, como por exemplo, fabricante, preço, dimensões, voltagem, etc. Portanto, devemos utilizar apropriadamente o conceito de abstração na análise de sistemas, representando nos sistemas que vamos criar apenas aquelas características que nos interessam dos objetos reais, isto é, as características relevantes para o sistema a ser projetado e implementado.

1.3.3 – Encapsulamento

Outro conceito importante na orientação a objetos é o **encapsulamento**. Ele permite que certas características ou propriedades dos objetos de uma classe **não possam ser vistas ou modificadas externamente,** ou seja, ocultam-se as características internas do objeto. Por exemplo, no caso da Cozinha, há diversas propriedades do Forno que fazem com que ele cozinhe o alimento, mas elas ficam ocultas para os demais objetos, não importando como ocorrem, mas apenas que o resultado final seja obtido, que é o cozimento do alimento. Se for substituído o forno, todos os demais objetos continuam intactos – apenas muda a forma de cozimento internamente à classe Forno. No contexto de uma linguagem OO, o encapsulamento protege os dados que estão dentro dos objetos, evitando que eles sejam alterados incorretamente. A única forma de alterar esses dados é por meio de funções dos próprios objetos, que são chamadas de operações ou métodos.

1.3.4 – Herança

Herança é um mecanismo que permite que características comuns a diversas classes sejam colocadas em uma classe base, ou superclasse. A partir de uma classe base, outras classes podem ser especificadas (as subclasses). Cada subclasse apresenta as características (estrutura e métodos) da superclasse e acrescenta a elas novas características ou comportamento. Dizemos que uma subclasse **herda todas as propriedades da superclasse e acrescenta suas características próprias e exclusivas**. As propriedades da superclasse não precisam ser repetidas em cada subclasse.

Por exemplo, quando pensamos em um forno, logo nos vem à mente sua utilidade geral, que é cozinhar alimentos. Várias subclasses de Forno adicionam características próprias aos diversos tipos de forno, como microondas, à gás, elétrico e à lenha, conforme ilustrado na Figura 4. Podemos criar diversos níveis de hierarquia de herança. Por exemplo, um forno pode ter vários sub-tipos, tais como microondas, à gás, elétrico e à lenha.

Outro recurso disponibilizado por algumas abordagens OO é a **herança múltipla**. Nesse caso, uma classe pode herdar de mais de uma classe, ou seja, uma subclasse possui duas ou mais super-classes, das quais herda todos os atributos e comportamento. Na Figura 3 apresenta-se o caso em que um Leitor da Biblioteca possui como superclasses UNIVERSIDADE FEDERAL DE SÃO CARLOS Estudante e Funcionário. Isso significa que ele herda os atributos de ambas as classes, além de ter um atributo próprio. Problemas podem ocorrer ao projetar sistemas OO usando herança, principalmente se existir comportamento com o mesmo nome em ambas as superclasses da classe com herança múltipla. Por exemplo, se tanto Estudante quanto Funcionário possuírem uma funcionalidade para “calcularDataDeDevolução”, cada qual retornando um valor diferente (por exemplo, dataDeHoje + 7 e dataDeHoje + 15), e se a classe Leitor não definir novamente essa funcionalidade, ficará a cargo do compilador saber qual dos dois comportamentos será executado (provavelmente será escolhida uma das alternativas, para o caso do programador esquecer de informar qual delas quer usar).

Outras características importantes da orientação a objetos, como polimorfismo, agregação, etc., serão exploradas nas unidades seguintes, na medida em que forem necessários para entendimento dos conceitos aos quais se relacionam.

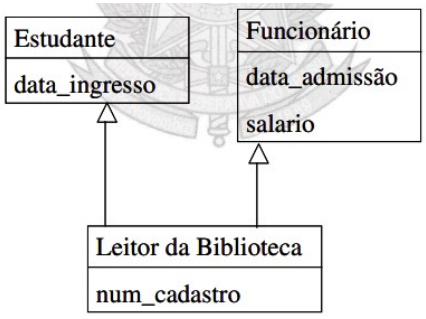

Figura 3 - Exemplo de Herança.

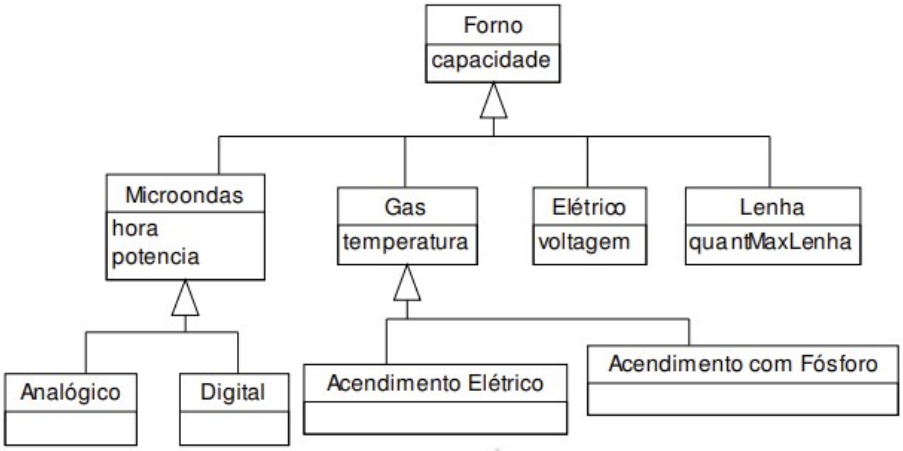

Figura 4 – Exemplo de Herança Múltipla.

2 – Modelos de Processo de Desenvolvimento de Software e o Processo Unificado

2.1 – Modelos de Processo de Software

Desenvolver software é geralmente uma tarefa complexa e sujeita a erros. O processo que inicia quando surge a necessidade de desenvolver um software para resolver um determinado problema e termina quando o software é implantado, pode tomar diversos rumos, dependendo de inúmeros fatores que ocorrem durante todo o processo. Por exemplo, o processo pode ser mais ou menos eficiente de acordo com a forma como as tarefas são distribuídas aos responsáveis pelo desenvolvimento, os artefatos que devem ser produzidos ao final e as decisões tomadas ao longo do desenvolvimento, entre outros.

Ao longo dos anos, empresas desenvolvedoras de software têm documentado seus processos, dando origem a diversos modelos de processo de software, também chamados de paradigmas de desenvolvimento de software. Cada modelo de processo de software fornece uma representação abstrata de um processo em particular, que pode ser seguido pelo desenvolvedor de software. Na verdade, ele representa uma tentativa de colocar ordem em uma atividade inerentemente caótica, que é a do desenvolvimento de software. Como exemplos de modelos podem-se citar: **Modelo em Cascata, Modelo de Prototipagem, Modelo Evolucionário, Desenvolvimento Baseado em Componentes e Modelo de Métodos formais**.

Muitos desses modelos surgiram para aprimorar outro modelo existente, adaptando-o a novas situações e ambientes de desenvolvimento. Por exemplo, o modelo em cascata estabelece que o desenvolvimento inicia com a coleta de requisitos do sistema e prossegue com a análise, projeto, codificação e finalmente teste do sistema, conforme ilustrado na Figura 5. Esse modelo apresenta **diversos problemas** em decorrência de sua sequência totalmente **linear** de atividades, o que **não corresponde** ao que ocorre **na realidade** de um projeto.

O problema mais grave ocorre quando mudanças nos requisitos são notadas em fases mais avançadas do processo. O retrabalho exigido para encaixar essas mudanças pode comprometer custos e cronogramas. O segundo problema está relacionado ao fato de que muitas vezes o usuário não consegue estabelecer os requisitos explicitamente no início do processo. Isso torna o modelo em cascata **ineficiente**, pois ele depende dos requisitos para prosseguir para as demais fases. Outro problema é que uma versão executável do sistema só será obtida no fim do processo e, portanto, o usuário não terá como implantar o sistema de forma incremental.

Por outro lado, o modelo em Cascata trouxe contribuições importantes para o processo de desenvolvimento de software: **imposição de disciplina, planejamento e gerenciamento**, além de fazer com que a implementação do produto seja postergada até que os objetivos tenham sido completamente entendidos.

O modelo de prototipagem surgiu para amenizar o segundo problema do modelo em Cascata, pois, com a criação de protótipos durante o processo, o cliente tem mais chances de antever os resultados e melhor definir seus requisitos. Protótipos são versões do programa que está sendo desenvolvido que são apresentadas ao cliente com algumas funcionalidades disponíveis. Ao decorrer do ciclo, novas funcionalidades são adicionadas ao protótipo até que seja feita a versão final do software. A Figura 6 ilustra o modelo de prototipagem. Embora resolva um problema do modelo em cascata, o modelo de prototipagem introduz outros problemas: o cliente não sabe que o software que ele obtém em cada ciclo não considerou, durante o desenvolvimento, a qualidade global e a manutenibilidade a longo prazo. É bastante comum que o desenvolvedor faça uma implementação de baixa qualidade (incompleta e com baixo desempenho), com o objetivo de produzir rapidamente um protótipo. Assim, o cliente precisa estar consciente de que o protótipo servirá apenas como um **mecanismo de definição dos requisitos**, podendo ser descartado no final do ciclo, quando então o produto final será desenvolvido de acordo com todos os requisitos de qualidade exigidos.

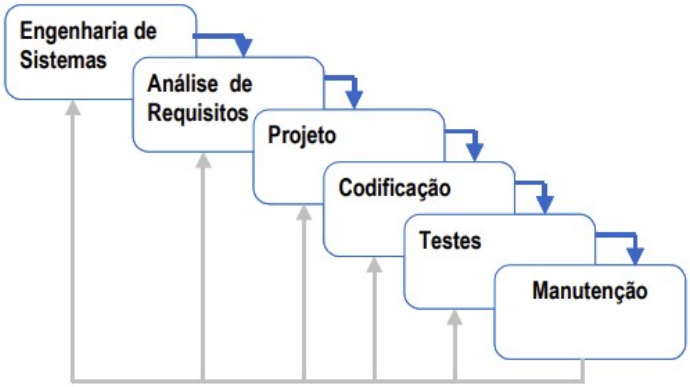

Figura 5 – Modelo em Cascata.

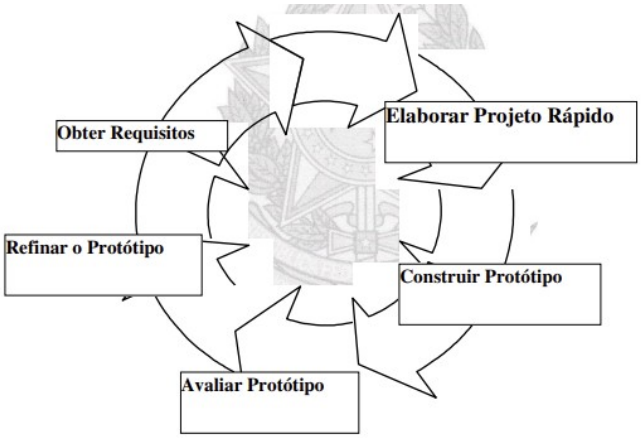

Figura 6 – Modelo de Prototipagem.

2.2 – Modelo Incremental

O modelo incremental possui características compostas a partir dos modelos de processo de software mencionados anteriormente: o Modelo em Cascata e o Modelo de Prototipagem. A idéia é conduzir **pequenos ciclos de desenvolvimento** para descobrir os requisitos do usuário, cada qual produzindo incrementos ao sistema, de forma que o sistema final seja composto desses incrementos. A Figura 7 ilustra o Modelo Incremental. Em geral, a versão inicial é o núcleo do produto (a parte mais importante). A evolução acontece quando **novas características são adicionadas à medida que são sugeridas pelo usuário**.

Este modelo é importante quando é difícil estabelecer a priori uma especificação detalhada dos requisitos, mas difere do modelo de prototipagem porque **produtos operacionais são produzidos a cada iteração**, ou seja, cada incremento resulta em produto que possui qualidade, **embora não atenda a todas as funções desejadas**. As novas versões podem ser planejadas para que os riscos técnicos possam ser administrados (por exemplo, de acordo com a disponibilidade de determinado hardware). Com o modelo incremental, tenta-se sanar os problemas dos modelos em cascata e de prototipagem. O desenvolvimento incremental resolve tanto o problema de **não se conhecer todos os requisitos a princípio**, pois pode-se identificá-los aos poucos durante os incrementos, quanto o problema do usuário ter que esperar até o fim do processo para ter o software executável, pois as diversas partes produzidas podem entrar em operação imediatamente, ou pelo menos servir para treinamento dos usuários e teste de sistema. Ao mesmo tempo, resolve o problema da qualidade dos protótipos produzidos, pois cada incremento é feito seguindo normas de qualidade.

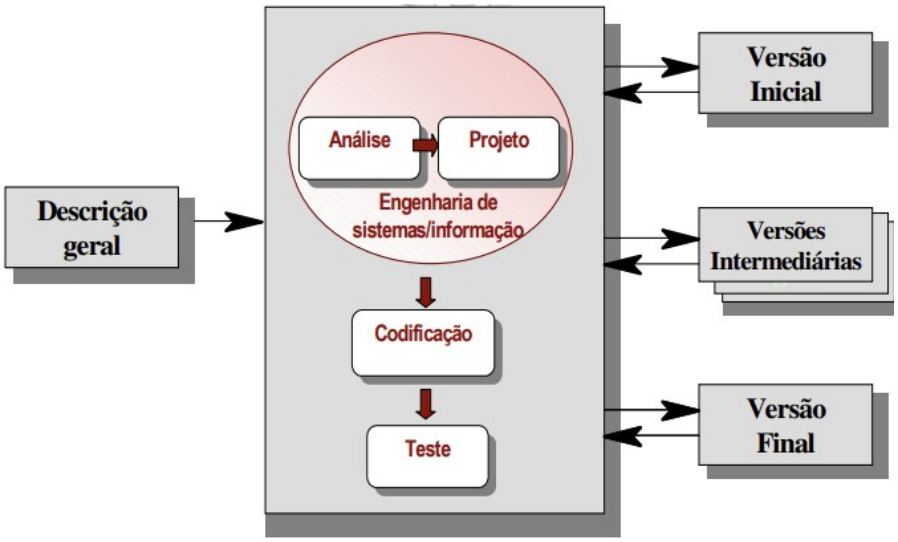

Figura 7 – Modelo Incremental

2.3 – O Processo Unificado (PU)

2.3.1 – Visão Geral

O **Processo Unificado (PU)** é um modelo de processo de software baseado no modelo incremental, visando à construção de software orientado a objetos. Ele usa como notação de apoio a UML (Unified Modeling Language) [Rational, 2000]. O RUP (Rational Unified Process) é um refinamento detalhado do PU, proposta pela empresa IBM Rational [Krutchen, 2000]. A idéia mais importante do PU é o desenvolvimento **"iterativo"**. O desenvolvimento de um software é dividido em vários "ciclos de iteração", cada qual produzindo um sistema **testado, integrado e executável**. Em cada ciclo ocorrem as atividades de **análise de requisitos, projeto, implementação e teste**, bem como a "integração" dos artefatos produzidos com os artefatos já existentes, conforme ilustrado na Figura 8.

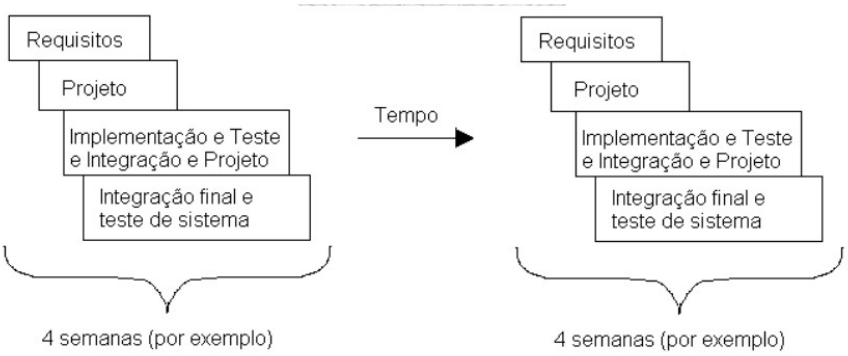

Figura 8 – Desenvolvimento Iterativo e Incremental [Larman, 2004].

No início do desenvolvimento, deve-se planejar quantos ciclos de desenvolvimento serão necessários para alcançar os objetivos do sistema, para que as partes mais importantes sejam priorizadas e alocadas nos primeiros ciclos. Portanto, é de vital importância no PU que a primeira iteração estabeleça os principais riscos e o escopo inicial do projeto, de acordo com a funcionalidade principal do sistema. Deve-se evitar correr o risco de descobrir, em iterações posteriores, que o projeto é inviável. Para isso, as partes mais complexas do sistema devem ser atacadas já no primeiro ciclo, pois são elas que apresentam maiores riscos de inviabilizar o projeto.

O tamanho de cada ciclo pode variar de uma empresa para outra e conforme o tamanho do sistema. Por exemplo, uma empresa pode desejar ciclos de 4 semanas, outra pode preferir 3 meses. Produtos entregues em um ciclo podem ser colocados imediatamente em operação, mas podem vir a ser substituídos por outros produtos mais completos em ciclos posteriores. É claro que isso deve ser muito bem planejado, pois, caso contrário, o esforço necessário para substituir um produto por outro pode vir a comprometer o sucesso do desenvolvimento incremental.

Um “caso de uso” (que será abordado mais adiante neste curso) é uma seqüência de ações executadas de forma colaborativa entre o sistema e um ou mais usuários, para produzir um ou mais resultados valiosos para os interessados no sistema. O PU é dirigido por casos de uso, pois os utiliza para dirigir todo o trabalho de desenvolvimento, desde a captação inicial e negociação dos requisitos até a aceitação do código (testes). Os casos de uso são centrais ao PU e outros métodos iterativos, pois:

- Os requisitos funcionais são registrados preferencialmente por meio deles;
- Eles ajudam a planejar as iterações;
- Eles podem conduzir o projeto;
- O teste é baseado neles.

No PU, a arquitetura do sistema em construção é o alicerce fundamental sobre o qual ele se erguerá. Entende-se por arquitetura a organização fundamental de todo o sistema, o que inclui elementos estáticos, dinâmicos, o modo como trabalham juntos e o estilo arquitetônico total que guia a organização do sistema.

A arquitetura também se refere a questões como desempenho, escalabilidade, reúso e restrições econômicas e tecnológicas. Assim, a arquitetura deve ser uma das preocupações da equipe de projeto.

A arquitetura, juntamente com os casos de uso, deve orientar a exploração de todos os aspectos do sistema. Os motivos que fazem com que a arquitetura seja importante no PU são:

- Melhor entendimento da visão global do sistema. Essa visão é importante e não pode ser deixada de lado em nenhum momento, para que o projeto **não se desvie do foco principal para o qual foi concebido**;
- Melhoria da organização do esforço de desenvolvimento: por exemplo, a modularização pode **diminuir a complexidade do sistema** por meio de módulos mais gerenciáveis por equipes paralelas de desenvolvimento;
- Aumento da possibilidade de reúso – camadas podem ser projetadas para que possam ser **reutilizadas em outros projetos**;
- Facilidade de evolução do sistema – um projeto com uma boa arquitetura pode ser **mais flexível e previsível** em relação a futuras mudanças;
- Guia para seleção e exploração dos casos de uso – uma boa arquitetura facilita a alocação dos casos de uso nos diversos ciclos de desenvolvimento.

2.3.2 – As Fases do PU

Cada um dos ciclos de desenvolvimento do PU é dividido em **quatro** fases: **concepção**, **elaboração, construção e transição**, conforme ilustrado na Figura 9. Marcos referenciais podem ser estabelecidos em um ou mais pontos, nos quais são tomadas decisões que permitem o prosseguimento normal do desenvolvimento ou podem exigir reajustes de cronograma e custos.

Na fase de **concepção** estabelece-se a viabilidade de implantação do sistema. Após definir o escopo do sistema, identificando as funcionalidades que pertencerão ou não a ele, pode-se fazer estimativas de custos e cronograma, além de identificar os potenciais riscos que devem ser gerenciados ao longo do projeto. Faz-se ainda o esboço da arquitetura do sistema, que servirá como alicerce para a sua construção.

Na fase de **elaboração** cria-se uma visão refinada do sistema, com a definição dos requisitos funcionais, detalhamento da arquitetura criada na fase anterior e gerenciamento contínuo dos riscos envolvidos. Estimativas realistas feitas nesta fase permitem preparar um plano para orientar a construção do sistema.

Na fase de **construção** o sistema é efetivamente desenvolvido e, em geral, tem condições de ser operado, mesmo que em ambiente de teste, pelos clientes.

Na fase de **transição** o sistema é entregue ao cliente para uso em produção. Testes são realizados e um ou mais **incrementos** do sistema são implantados. Um incremento é a diferença entre entregas feitas em iterações uma seguida à outra. Defeitos são corrigidos, se necessário.

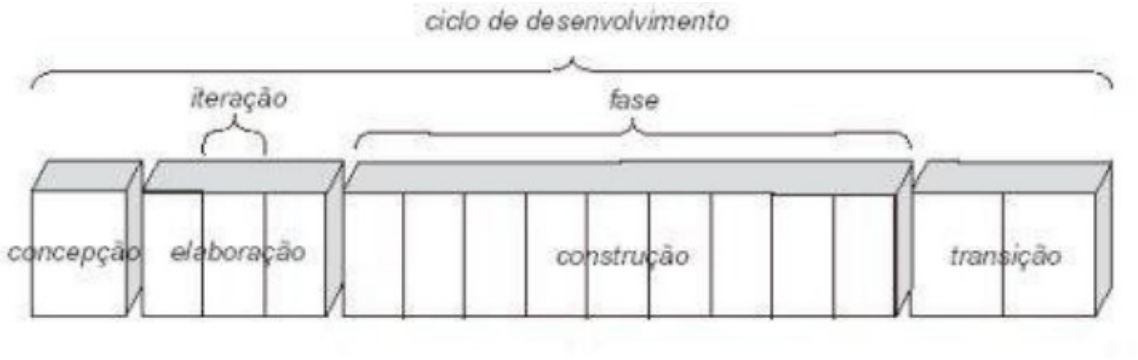

Figura 9 – Fases de um ciclo de desenvolvimento no PU [Larman, 2004].

2.3.3 – As Disciplinas do PU

Se analisarmos as fases do PU, podemos ter a impressão de que cada ciclo de iteração comporta-se como o modelo Cascata, visto na seção 2.1. Mas isso não é verdade: paralelamente às fases do PU, atividades de trabalho, denominadas disciplinas do PU, são realizadas a qualquer momento durante o ciclo de desenvolvimento. Conforme ilustrado na Figura 10, as disciplinas entrecortam ortogonalmente todas as fases do PU, podendo ter maior ênfase durante certas fases e menor ênfase em outras, mas podendo ocorrer em qualquer uma delas. Considerando a disciplina "Projeto", nota-se no exemplo da Figura 10 que ela tem maior ênfase no fim da fase de elaboração e no início da fase de construção.

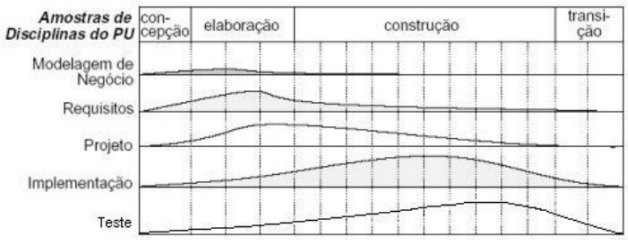

Figura 10 – Disciplinas e Fases do PU [Larman, 2004].

Como exemplo de disciplinas pode-se mencionar como as mais importantes: Modelagem de Negócio, Requisitos, Projeto, Implementação, Teste, Implantação, Gestão de Configuração e Mudanças, Gestão de Projeto e Ambiente. A ênfase ou carga de uma disciplina em cada fase pode variar de projeto para projeto. Algumas disciplinas podem ter grande parte de sua carga associada a uma das fases como a disciplina "Implementação" tem grande ênfase na fase de Construção, enquanto que "Requisitos" é uma disciplina com pouca ênfase nessa fase. Outras disciplinas são importantes em várias fases, portanto, a ênfase é distribuída ao longo de todas elas, como a disciplina Teste, cuja distribuição é praticamente igual nas fases de elaboração, construção e transição.

2.3.4 – Os Artefatos do PU

Cada uma das disciplinas do PU pode gerar um ou mais artefatos, que devem ser controlados e administrados corretamente durante o desenvolvimento do sistema. Artefatos podem ser quaisquer documentos produzidos durante o desenvolvimento, tais como modelos, diagramas, documento de especificação de requisitos, código fonte, código executável, planos de teste etc. Muitos dos artefatos são opcionais, sendo produzidos de acordo com as necessidades específicas de cada projeto.

A Tabela 1, baseada na sugestão de Larman [2004], mostra exemplos de alguns dos artefatos correspondentes a cada uma das disciplinas do PU, bem como as fases nas quais eles são produzidos inicialmente ou refinados. Há documentos produzidos que não se tornam artefatos e são descartados. Geralmente são documentos intermediários que apóiam alguma atividade ou a geração de outros documentos, como formulários utilizados para entrevistas e planilhas para cálculos e estimativas.

Tabela 1 – Exemplos de Artefatos do PU (p – produzir, r – refinar).

Disciplina	Artefato *Iteração* →	Concepção C_1	Elaboração $E_1...E_n$	Construção $C_1...C_n$	Transição $T_1...T_n$
Modelagem de Negócio	Modelo Conceitual		p		
Requisitos	Diagrama de Casos de Uso	P	r		
	Casos de Uso Textuais	P	r		
	Diagrama de Seqüência do Sistema	P	r		
	Contratos para operações	P	r		
	Glossário	P	r		
Projeto	Diagrama de Classes		p	r	
	Diagrama de Colaboração		p	r	
	Diagrama de Pacotes		p	r	
	Documento de Arquitetura do Software		p		
Implementação	Código fonte			p	r

Referências [Alexander 77] Christopher Alexander et. al., A Pattern Language, Oxford University Press, New York, 1977.

[Alexander 79] Christopher Alexander, The Timeless Way of Building, Oxford University Press, New York, 1979.

[Appleton 97] Appleton, Brad. Patterns and Software: Essential Concepts and Terminology, disponível na WWW na URL: http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html

[Bauer, 2004] Bauer, Christian; King, Gavin. Hibernate in Action, Manning Publications. [Beck 87] Beck, Kent; Cunningham, Ward. Using Pattern Languages for Object-Oriented Programs, Technical Report nº CR-87-43, 1987, disponível na WWW na URL: http://c2.com/doc/oopsla87.html

[Booch, 1995] Object Solutions : Managing the Object-Oriented Project (Addison-Wesley Object Technology Series by Grady Booch , Pearson Education; 1st edition (October 12, 1995)

[Buschmann 96] Buschmann, F. et at. A System of Patterns, Wiley, 1996.

[Coad 92] Coad, Peter. Object-Oriented Patterns. Communications of the ACM, V. 35, nº9, p. 152-159, setembro 1992.

[Coad 95] Coad, P.; North, D.; Mayfield, M. Object Models: Strategies, Patterns and Applications, Yourdon Press, 1995.

[Coleman et al, 1994] Coleman, Derek et al. Object Oriented Development - the Fusion Method, Prentice Hall, 1994.

[Coplien 92] Coplien, J.O. Advanced C++ Programming Styles and Idioms. Reading-MA, Addison-Wesley, 1992.

[Coplien 95] Coplien, J.; Schmidt, D. (eds.) Pattern Languages of Program Design, Reading-MA, Addison-Wesley, 1995.

[Deitel, 2002] Deitel, Harvey M; Deitel Paul J. JAVA – como programar, Editora Bookman, 4a Edição.

[Gamma 95] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. Design Patterns - Elements of Reusable Object-Oriented Software. Reading-MA, Addison-Wesley, 1995.

[Goodwill, 2002] Goodwill,J. Martering Jakarta Struts, Wiley Publishing, Inc.

[Krutchen, 2000] Krutchen, Philippe. The Rational Unified Process, An Introduction, Second Edition, Addison Wesley, 2000.]

[Larman, 2004] Larman, Craig. Utilizando UML e Padrões, 2a edição, Bookman, 2004.