UNIVERSIDADE DA REGIÃO DE JOINVILLE – UNIVILLE

DEPARTAMENTO DE INFORMÁTICA

DESENVOLVIMENTO DE UM *FRAMEWORK* PARA CONSTRUÇÃO DE APLICAÇÕES COMERCIAIS

KAMYLA ESTUQUI PARRADO

PROFESSOR WALTER SILVESTRE COAN

Joinville – SC

2008

KAMYLA ESTUQUI PARRADO

DESENVOLVIMENTO DE UM *FRAMEWORK* PARA CONSTRUÇÃO DE APLICAÇÕES COMERCIAIS

Trabalho de Conclusão de Estágio apresentado ao curso de Sistemas de Informação da Universidade da Região de Joinville – UNIVILLE – como requisito parcial para obtenção do grau de Bacharel em Sistemas de Informação, sob orientação do professor Walter Silvestre Coan.

Joinville – SC

2008

FOLHA DE APROVAÇÃO

A aluna Kamyla Estuqui Parrado, regularmente matriculada na 4ª série do curso de Bacharelado em Sistemas de Informação, apresentou e defendeu o Trabalho de Conclusão de Estágio, obtendo da Banca Examinadora a média final ____ (_____), tendo sido considerada aprovada.

Joinville, _____ de _____ de 2008.

_____	_____	_____
Prof. Walter Silvestre Coan	Prof. Daniel Maniglia Amancio da Silva	Prof. Paulo Marcondes Bousfield

UNIVERSIDADE DA REGIÃO DE JOINVILLE – UNIVILLE
TRABALHO DE CONCLUSÃO DE ESTÁGIO CURRICULAR SUPERVISIONADO
AVALIAÇÃO DO ESTAGIÁRIO PELA EMPRESA

Nome da Estagiária: Kamyla Estuqui Parrado

QUADRO I

a) AVALIAÇÃO NOS ASPECTOS PROFISSIONAIS	**Pontos**
1 – QUALIDADE DO TRABALHO – Considerando o possível	
2 – ENGENHOSIDADE – Capacidade de sugerir, projetar, executar modificações ou inovações	
3 – CONHECIMENTO – Demonstrado no desenvolvimento das atividades programadas	
4 – CUMPRIMENTO DAS TAREFAS – Considerar o volume de atividades dentro do padrão razoável	
5 – ESPÍRITO INQUISITIVO – Disposição demonstrada para aprender	
6 – INICIATIVA – No desenvolvimento das atividades	
SOMA	

Pontuação para o Quadro I e II
Sofrível – 1 ponto, Regular – 2 pontos, Bom – 3 pontos, Muito Bom – 4 pontos, Excelente – 5 pontos.

QUADRO II

b) AVALIAÇÃO DOS ASPECTOS HUMANOS	**Pontos**
1 – ASSIDUIDADE – Cumprimento do horário e ausência de faltas	
2 – DISCIPLINA – Observância das normas internas da Empresa	
3 – SOCIABILIDADE – Facilidade de se integrar com os outros no ambiente de trabalho	
4 – COOPERAÇÃO – Disposição para cooperar com os demais para atender as atividades	
5 – SENSO DE RESPONSABILIDADE – Zelo pelo material, equipamentos e bens da empresa	
SOMA	

c) AVALIAÇÃO FINAL	**Pontos**
SOMA do Quadro I multiplicada por 7	
SOMA do Quadro II multiplicada por 3	
SOMA TOTAL	

LIMITES PARA CONCEITUAÇÃO
De 57 a 101 – SOFRÍVEL
De 102 a 146 – REGULAR
De 148 a 194 – BOM
De 195 a 240 – MUITO BOM
De 241 a 285 – EXCELENTE

Nome da Empresa: Clean Processamento de Dados
Representada pelo Supervisor: Roberto Pereira

CONCEITO CONFORME SOMA TOTAL	**Rubrica do Supervisor da Empresa**	
	Local: Joinville Data:	Carimbo da Empresa

AGRADECIMENTOS

Agradeço a Deus por ter me dado força e coragem para prosseguir, mesmo diante de tantos desafios. À minha família, por sempre estar ao meu lado, me apoiando e incentivando. Aos meus colegas de turma pela troca de experiências. Aos meus professores, mestres que me ensinaram muitas lições, responsáveis pela minha formação acadêmica e pelo conhecimento que tenho hoje. E, principalmente, agradeço ao meu orientador específico, Walter S. Coan, pelo apoio, incentivo, paciência e excelente orientação durante todos esses meses.

EPÍGRAFE

“A combinação de pensar e fazer produz a grandeza. O pensamento e a clara visão tornam as ações eficazes, transformando nossos sonhos em realidade. O pensamento não substitui a ação, nem a ação substitui o pensamento, pois as duas coisas são complementares. O esforço necessário para conseguir algo, está justamente em superar nossos limites, procurando conciliar pensamentos e ações. Para aqueles que fazem este esforço e se superam, qualquer coisa é possível”. (Cândida Inthurn)

RESUMO

Este trabalho de conclusão de estágio consiste na análise, projeto e desenvolvimento de um *framework*. O objetivo deste *framework* é atender a demanda de desenvolvimento de software da empresa *Clean* Informática Ltda. O foco principal da empresa é o desenvolvimento e comercialização de sistemas para empresas prestadoras de serviços. No entanto, a empresa também possui um sistema, *Agines Network*, voltado para indústria, comércio e ferramentarias. O principal agente motivador para o desenvolvimento do *framework* é a grande dificuldade para desenvolver e manter o sistema *Agines Network*. Além disso, as novas estratégias da empresa visam o desenvolvimento de soluções reutilizáveis. Para atender as estratégias da empresa, aumentar a produtividade do setor de desenvolvimento e aprimorar a qualidade dos produtos da empresa, será desenvolvido um *framework* que servirá como base para o desenvolvimento dos produtos da empresa.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

INTRODUÇÃO

A reutilização de software tem se tornado um tema muito comentado nos últimos anos. Diversas técnicas para reutilização, não somente de implementação, mas também de análise e *design* têm surgido.

Atualmente, os padrões de projeto (*design patterns*) e a análise e projeto orientado a objetos, têm sido os principais representantes das técnicas de reutilização. Os *frameworks* representam uma forma eficaz de aplicar os conceitos propostos pela orientação a objetos e pelos padrões de projeto. O resultado do desenvolvimento e utilização de *frameworks* é a reutilização de análise, design e implementação.

O objetivo deste projeto é, através da construção de um *framework*, demonstrar a importância da aplicação das técnicas propostas pela engenharia de software ao desenvolvimento de sistemas baseados em computador. Além disso, este projeto está focado na reutilização, tanto de componentes de software, quanto de análise e *design*. Outros pontos chaves são os requisitos não-funcionais de um sistema, tais como manutenibilidade, usabilidade e desempenho.

Este projeto está dividido em três partes: projeto de estágio, fundamentação teórica e descrição das atividades práticas. O projeto de estágio descreve os objetivos (geral e específicos), o problema e a solução proposta, bem como a metodologia utilizada para o desenvolvimento do projeto e o cronograma de atividades do estágio.

A fundamentação teórica trata dos conceitos teóricos necessários para o desenvolvimento e utilização de *frameworks*. As seções deste capítulo descrevem conceitos sobre engenharia, arquitetura, componentes e testes de software, princípios de orientação a objetos, padrões de projeto e *frameworks*. Além disso, também conceitua o *Unified Process* e a UML. Apresenta conceitos sobre as linguagens de programação *Java* e *Flex*, e sobre o paradigma da *Web* 2.0 e das *Rich Internet Applications*. E, finaliza, com os conceitos relacionados à persistência de dados.

Na terceira parte são descritas as atividades práticas realizadas para a execução deste projeto, os artefatos e resultados obtidos em cada etapa. As atividades práticas se iniciam com o levantamento de requisitos, onde são identificados os primeiros cenários arquiteturais do *framework*. Através dos cenários arquiteturais são definidos os atores e casos de uso. Após o levantamento de requisitos, é feita uma análise sobre as arquiteturas candidatas para o desenvolvimento do *framework*. A etapa seguinte é o *design*, onde são descritas mais detalhadamente as soluções utilizadas para construir o *framework*. Durante a fase de

implementação, as soluções descritas na etapa de *design* são transformadas em artefatos de software executáveis. Após a implementação, são planejados e executados os testes necessários para a validação das funcionalidades desenvolvidas durante a implementação. Por fim, são descritos os processos e ferramentas necessários para implantar o *framework* e os processos necessários para manter e documentar o *framework*.

1. PROJETO DE ESTÁGIO

1.1. Identificação do estágio

1.1.1. Dados de Identificação do Aluno

Nome: Kamyla Estuqui Parrado.
Curso: Bacharelado em Sistemas de Informação.
Endereço: Rua Max Lebowski, 643 – Rocio Grande – São Francisco do Sul – SC.

1.1.2. Dados de Identificação da Empresa

Denominação: *Clean* Informática Ltda.
Ramo de Atividade: Desenvolvimento e comercialização de softwares.
Endereço: Rua Iririú, 1152 – Iririú – Joinville – SC.
Fone: (47) 3437-3312.

1.1.3. Dados dos Responsáveis pelo Estágio

Orientador de Classe: Paulo Marcondes Bolsfield.
Orientador Específico: Walter Silvestre Coan.
Supervisor no Campo de Estágio: Roberto Pereira.

1.2. Tema

Desenvolvimento de um *framework* que ofereça suporte à construção de aplicações comerciais, voltadas as mais diversas áreas.

1.3. Assunto

A constante evolução na área de Tecnologia da Informação (TI), aliada ao ritmo de trabalho acelerado das empresas que atuam nesta área, fez com que surgisse uma demanda

pelo desenvolvimento de software de forma ágil e eficaz. Essa demanda surgiu da necessidade de aumentar a produtividade, reduzir custos decorrentes do desenvolvimento de software e aumentar a qualidade dos produtos e serviços oferecidos.

Nesse contexto, as linguagens de programação e as técnicas de desenvolvimento de software evoluíram, fazendo com que surgissem novas ferramentas e técnicas para auxiliar o processo de desenvolvimento de software. Uma dessas técnicas é conhecida como *framework*.

Booch, Jacobson e Rumbaugh (2005, 390), definem *framework* como "um padrão de arquitetura que fornece um *template* extensível para aplicações dentro de um domínio, abrangendo um conjunto de mecanismos que trabalham em conjunto para resolver um problema básico de um domínio comum".

De forma geral, um *framework* pode ser considerado como uma técnica da Orientação a Objetos, voltada a reutilização, que se beneficia de três características das linguagens de programação orientadas a objetos: abstração, polimorfismo e herança. (FERREIRA, 2008).

1.4. Problema

Atualmente, o software *Agines Network*, desenvolvido pela empresa *Clean* Informática Ltda, apresenta uma série de problemas como os seguintes:

- Falta de padronização no código;
- Código duplicado em diversas partes do sistema;
- Dificuldade de manutenção;
- Dificuldade de implementação de novas funcionalidades;
- Ausência de componentes que realizem tarefas básicas do sistema, entre outros.

Os problemas citados acima estão todos inter-relacionados, uma vez que a falta de padronização no código e de componentes para realizar tarefas básicas do sistema, geram código duplicado. Este, por sua vez, dificulta a manutenção do sistema e a implementação de novas funcionalidades. Conseqüentemente, a qualidade do software diminui, gerando as seguintes deficiências:

- **Desempenho abaixo do esperado**: devido à inexistência de uma arquitetura específica e padrões de desenvolvimento, ocorre a perda de performance;
- **Falta de escalabilidade**: o sistema não esta preparado para atender a demanda de novas funcionalidades solicitadas pelos clientes ou pelo mercado;

- **Re-trabalho**: para alterar ou criar uma funcionalidade é necessário modificar várias partes do sistema que já estavam prontas, testadas e funcionando.

Todos os fatores descritos acima contribuem para a inexistência de uma arquitetura adequada que atenda as necessidades de desenvolvimento de software da empresa.

1.5. Objetivos

1.5.1. Objetivo geral

Desenvolver um *framework* para construção de aplicações comerciais.

1.5.2. Objetivos específicos

i. Definir a arquitetura de software a ser utilizada;
ii. Identificar os componentes do *framework*;
iii. Analisar as alternativas disponíveis para construir os componentes do *framework*;
iv. Projetar arquitetura do *framework*;
v. Priorizar e construir os componentes do *framework*;
vi. Efetuar testes nos componentes do *framework*;
vii. Gerar documentação adequada para a manutenção e utilização do *framework*;
viii. Implantar o *framework* na empresa.

1.6. Justificativa

Atualmente, as organizações envolvidas com a Tecnologia da Informação, precisam manter um alto nível de qualidade em seus produtos e serviços, bem como aprimorar o processo de desenvolvimento de software para aumentar a produtividade. Para chegar a esses objetivos, no entanto, é necessário que um software possua algumas características. Essas características são conhecidas, de acordo com Scott (2003), dentro da engenharia de software, como *"bilidades"*: usabilidade, escalabilidade, manutenibilidade, confiabilidade, entre outras.

Essas características podem ser alcançadas através do desenvolvimento de um *framework* que possa centralizar as rotinas mais comuns de um sistema, possibilitando ser reutilizado em outros projetos. Os *frameworks* não buscam apenas reutilizar componentes de

software, mas subsistemas, análise e design, aumentando assim o grau de reutilização e contribuindo para aumentar a qualidade do software. (FERREIRA, 2008)

Segundo Fayad (1999 *apud* FERREIRA, 2008), a utilização de *frameworks* apresenta os seguintes benefícios:

- Melhora a modularização, encapsulando detalhes de implementação através de interfaces estáveis;
- Aumenta a reutilização, definindo componentes genéricos que podem ser utilizados para a criação de novos sistemas;
- Extensibilidade, favorecida pelo uso de métodos *hooks* [1] que permitem que as aplicações estendam interfaces estáveis;
- Inversão de controle, o código do desenvolvedor é chamado pelo código do *framework*. Dessa forma, o *framework* controla a estrutura e o fluxo de execução dos programas.

Larman (2002), afirma que *frameworks* fornecem um grau elevado de reutilização, conseqüentemente, se uma organização tem interesse em aumentar o nível de reutilização de software, então é crítico enfatizar a criação de *frameworks*. A partir do momento em que um sistema passa a utilizar uma solução que já foi testada e aprovada, tanto pelos desenvolvedores quanto pelos clientes, este passa a ser um produto com qualidade superior.

1.7. Metodologia

Para este projeto serão utilizadas as seguintes metodologias:

- Pesquisa bibliográfica;
- Entrevistas;
- UP (*Unified Process*);
- UML (*Unified Modeling Language*)

O Processo Unificado (*Unified Process* – UP) é dirigido por casos de uso, centrado na arquitetura e iterativo e incremental. Esta metodologia de desenvolvimento de sistemas é definida por Scott (2003, p.19) como "um conjunto de atividades executadas para transformar um conjunto de requisitos do cliente em um sistema de software". Porém, Scott (2003) também afirma que o "Processo Unificado é uma estrutura genérica de processo que pode ser

[1] Métodos *hook*, são métodos com uma implementação padrão que podem ser redefinidas nas subclasses que estendem a classe que contém estes métodos.

customizado adicionando-se ou removendo-se atividades com base nas necessidades e nos recursos disponíveis para um projeto".

De acordo com Waslawick (2004), o UP comporta as etapas de estudo de viabilidade, análise de requisitos, análise de domínio, e projeto em múltiplas camadas. No entanto, essas etapas aparecem no UP organizadas em quatro fases: concepção, elaboração, construção e transição.

Dentro do UP, "cinco *workflows* atravessam o conjunto das quatro fases: requisitos, análise, projeto, implementação e teste. Cada *workflow* é um conjunto de atividades que vários membros do projeto executam." (SCOTT, 2003, p.29).

Scott (2003) afirma que o Processo Unificado faz uso extensivo da *Unified Modeling Language* (UML). Na visão desse autor, a UML foi projetada para auxiliar aqueles que participam da atividade de desenvolvimento de software a construir modelos que permitam visualizar o sistema. A UML também ajuda a especificar a estrutura e o comportamento do sistema, construí-lo e documentar as decisões tomadas durante o processo.

1.8. Cronograma

O cronograma apresenta as atividades realizadas durante o estágio. As células de cor cinza claro do cronograma representam o tempo previsto para realização das atividades. As células de cor preta representam o tempo que realmente foi necessário para realizar as atividades. As células de cor cinza escuro representam atividades pendentes ou parcialmente concluídas.

O cronograma apresenta as atividades distribuídas entre os meses de Fevereiro a Dezembro de 2008. Várias atividades foram realizadas em um mesmo mês e, algumas atividades foram realizadas simultaneamente.

Tabela 1 – Cronograma de atividades do estágio

Atividade	**Jan**	**Fev**	**Mar**	**Abr**	**Maio**	**Jun**
Definir projeto de estágio						
Desenvolver fundamentação teórica						

	Jul	**Ago**	**Set**	**Out**	**Nov**	**Dez**
Identificar cenários arquiteturais						

Descrever cenários arquiteturais						
Analisar cenários arquiteturais						
Identificar componentes e casos de uso do *framework*						
Priorizar casos de uso						
Modelar casos de uso						
Analisar as alternativas disponíveis para construir os componentes do *framework*						
Definir arquiteturas de software para o *framework*						
Projetar arquitetura do *framework*						
Implementar *framework*						
Testar *framework*						
Apresentar projeto a banca examinadora						

Fonte: Kamyla Estuqui Parrado (2008).

2. FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os principais conceitos relacionados ao desenvolvimento de *frameworks*, incluindo conceitos sobre engenharia, arquitetura, componentes e testes de software, padrões de projeto, *frameworks*, linguagens de programação, UML, *Unified Process* etc. Além disso, são apresentados conceitos relacionados à *Web* 2.0 e RIA, pois o *framework* que será construído destina-se ao desenvolvimento de aplicações comerciais voltadas ao ambiente *web*. Este capítulo também servirá como base para o desenvolvimento da parte prática deste projeto.

2.1. Teoria geral dos sistemas

Os sistemas estão presentes em todos os setores da sociedade. Existem sistemas de saúde pública, sistemas bancário, sistemas de prevenção de erros, entre outros. Devido à abrangência do termo sistema, este pode ser analisado a partir de diversos contextos.

Stair (1996, p.6) define um sistema como "um conjunto de elementos ou componentes que interagem para se atingir objetivos. Os próprios elementos e as relações entre eles determinam como o sistema trabalha".

O biólogo alemão *Ludwig Von Bertalanfy* é considerado o pai da Teoria Geral dos Sistemas (TGS). Este cientista foi o primeiro a propor que a teoria geral dos sistemas fosse aplicada a outras áreas além da biologia.

De acordo com Chiavenato (2000, p.543), "a TGS não busca solucionar problemas ou tentar soluções práticas; mas sim produzir teorias e formulações conceituais que possam criar condições de aplicações na realidade empírica". Segundo este autor, a TGS se baseia em três premissas:

1. Os sistemas existem dentro dos sistemas, ou seja, cada sistema é composto por subsistemas que, ao mesmo tempo, fazem parte de um sistema maior, definido por Chiavenato (2000) como supra-sistema;
2. Os sistemas são abertos, ou seja, existe troca de energia e informação entre os sistemas e o ambiente;
3. As funções de um sistema dependem de sua estrutura, ou seja, "cada sistema tem um objetivo ou finalidade, o que constitui seu papel no intercâmbio com outros sistemas dentro do ambiente" (CHIAVENATO, 2000, p.543).

2.1.1. Sistemas de informação

Um sistema de informação é um tipo especializado de sistema, utilizado pelas organizações como uma forma de controlar os processos internos, auxiliar na tomada de decisão, etc.

Laudon e Laudon (2001, p.40), definem os sistemas de informação como

> Um conjunto de componentes inter-relacionados que coleta (ou recupera), processa, armazena e distribui informação para dar suporte à tomada de decisão e ao controle da organização. Além de apoiar, coordenar e controlar a tomada de decisão, os sistemas de informação também podem ajudar os gerentes e trabalhadores a analisar problemas, visualizar assuntos complexos e criar novos produtos.

Uma definição complementar a esta é a de Bezerra (2002). Para esse autor, sistemas de informação são uma combinação de pessoas, dados, processos, redes de comunicação e tecnologia. Esses elementos interagem entre si para dar suporte ao processo de negócio de uma organização.

Os sistemas de informação têm como objetivo auxiliar os gerentes, e outros funcionários das organizações a tomar decisões, controlar o fluxo de trabalho, visualizar e resolver problemas. Essas tarefas são realizas pelos sistemas de informação através de três atividades: entrada, processamento e saída.

A entrada se refere à coleta de dados que, durante o processamento, são convertidos em informação útil a organização. A saída se refere ao envio da informação, obtida durante o processamento, às pessoas e atividades que utilizarão esta informação. O processamento se refere à transformação dos dados de entrada em informação útil a organização.

Além dessas três atividades, existe a realimentação, ou *feedback*, definida por Laudon e Laudon (1999, p.4) como "a saída que retorna aos membros adequados da organização para ajudá-los a refinar ou corrigir os dados de entrada". Para Andrade, Audy e Cidral (2005), o termo *feedback* está relacionado a controle. E, tem como objetivo, fiscalizar as informações dentro do sistema e alertar sobre a necessidade de ação, quando os resultados obtidos não correspondem ao plano original.

Os sistemas de informação assumem três papéis fundamentais dentro de qualquer organização:

- Suporte aos processos e operações;
- Suporte a tomada de decisão;
- Suporte às estratégias de vantagem competitiva.

2.1.2. Componentes dos sistemas de informação

A maneira mais adequada de tratar os sistemas de informação é através da abordagem sociotécnica, pois, esta proporciona uma visão integrada das dimensões humana, organizacional e tecnológica.

A dimensão organizacional diz respeito a "empresas formais que consistem em unidades especializadas com uma divisão nítida de mão-de-obra e especialistas empregados e treinados para diferentes funções profissionais" (LAUDON e LAUDON, 1999, p.6). As organizações moldam os sistemas de informação, uma vez que estes precisam se ajustar as necessidades de cada organização. A dimensão humana se refere aos usuários finais e aos especialistas em sistemas de informação. A dimensão tecnológica é definida por Laudon e Laudon (1999, p.6) como o "meio pelo qual os dados são transformados e organizados para uso das pessoas".

A Figura 1 demonstra a relação existente entre estes componentes e as atividades realizadas pelos sistemas de informação.

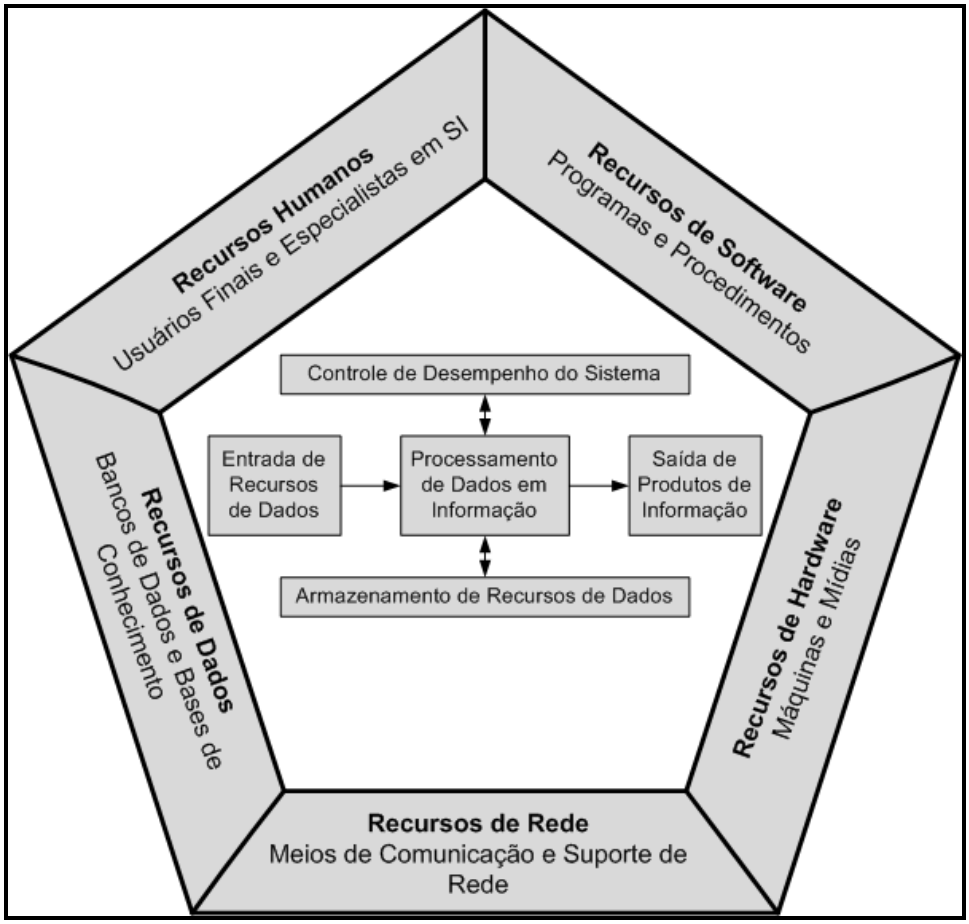

Figura 1 – Componentes dos sistemas de informação
Fonte: O'Brien (2003)

Levando em consideração as dimensões citadas anteriormente, pode-se afirmar que os sistemas de informação são compostos por cinco elementos:

- **Recursos humanos**: todas as pessoas envolvidas no processo de utilização e desenvolvimento dos sistemas de informação.
- **Recursos de hardware**: computadores e periféricos, máquinas e mídias em geral;
- **Recursos de software**: instruções utilizadas pelos recursos de hardware para manipular dados e informações. Alguns exemplos são: sistemas operacionais, planilhas eletrônicas, procedimentos para entrada de dados, etc.
- **Recursos de dados**: fatos armazenados nos recursos de hardware, que são transformados em informações e apresentados aos recursos humanos. Envolvem também os meios necessários para armazenar os dados. Exemplos de recursos de dados são: descrição de produtos, cadastro de clientes, banco de dados de estoque, etc.
- **Recursos de redes**: envolvem os meios necessários para que diversos recursos de hardware possam se comunicar, permitindo aos dados e informações serem transferidos de um lugar para outro ou compartilhados.

2.1.3. Tipos de sistemas

Os sistemas podem ser classificados de maneiras diferentes, baseando-se em diversos critérios. Um dos critérios utilizados está relacionado à topologia dos sistemas Neste sentido Reynolds e Stair (2002) propõem a seguinte classificação:

- **Simples x complexo**: sistemas simples possuem poucos componentes e as relações existentes entre esses componentes são descomplicadas. Por outro lado, sistemas complexos, normalmente possuem muitos componentes que são altamente interconectados.
- **Aberto x fechado**: sistemas abertos trocam informações com o ambiente em que estão inseridos, já os sistemas fechado não se comunicam com o ambiente externo. Segundo Andrade, Audy e Cidral (2005), no ambiente organizacional não existem sistemas fechados, costuma-se dizer que o sistema é parcialmente fechado ou semi-aberto.
- **Estável x dinâmico**: em sistemas estáveis, as mudanças que ocorrem no ambiente externo praticamente não causam mudanças no sistema. Já os sistemas dinâmicos sofrem mudanças rápidas e constantes, de acordo com o que ocorre no ambiente externo.

- **Adaptáveis x não-adaptáveis**: sistemas adaptáveis monitoram o ambiente "e recebem modificações em resposta às mudanças no ambiente. Um sistema não adaptável é o que não muda com um ambiente mutável" (STAIR, 1996, p.8).
- **Permanente x temporário**: sistemas permanentes costumam ser utilizados por um longo período de tempo, por exemplo, mais de dez anos, enquanto os sistemas temporários costumam durar menos de um mês.

Além dos tipos citados acima, Andrade, Audy e Cidral (2005) acrescentam os seguintes tipos:

- **Concretos x abstratos**: "sistemas concretos são formados por equipamentos, máquinas, pessoas e, de um modo geral, de objetos e artefatos reais" (ANDRADE, AUDY e CIDRAL, 2005, p.35). Já os sistemas abstratos podem ser exemplificados como teorias, conceitos ou hipóteses.
- **Naturais x artificiais**: sistemas naturais "têm suas origens na origem do universo, sendo resultados das forças e dos processos que caracterizam tal universo" (ANDRADE, AUDY e CIDRAL, 2005, p.35). Os sistemas artificiais são aqueles criados pelo homem, sistemas de informação baseados em computador são exemplos desse tipo de sistema.

Os sistemas de informação auxiliam as organizações a realizar suas atividades de maneira eficiente. Para que isso ocorra, no entanto, é necessário que os sistemas atendam aos diversos níveis hierárquicos de uma organização. Por isso, a classificação mais aceita para os sistemas de informação, é realizada utilizando como critério a finalidade principal de uso do sistema e o nível organizacional. Andrade, Audy e Cidral (2005) propõem a seguinte classificação:

- **Sistemas de processamento de transações**: executam e registram as transações que fazem parte da rotina e do processo de negócio das organizações.
- **Sistemas de informação gerencial**: "sintetizam, registram e relatam a situação em que se encontram as operações da organização" (ANDRADE, AUDY e CIDRAL, 2005, p.119). Sistemas desse tipo têm como público alvo os gerentes táticos das organizações e, têm como objetivo principal, apresentar indicadores de desempenho no formato de relatórios.
- **Sistemas de apoio à decisão**: auxiliam gerentes a tomar decisões com base em dados obtidos dos sistemas de processamento de transações, sistemas de informação gerencial e fontes externas. Esses sistemas também fornecem ferramentas para realização de análises e simulações do impacto de diferentes decisões.

- **Sistemas de informação executivo**: auxiliam aos executivos do nível estratégico a tomar decisões. Eles fornecem ambientes computacionais e de comunicação que permitem fácil acesso aos dados. Esse tipo de sistemas, normalmente, não tem por objetivo resolver problemas, mas sim, disponibilizar ferramentas que auxiliem os executivos a identificar problemas e oportunidades, decidir por alternativas de atuação, etc.

Para Laudon e Laudon (1999), os sistemas de informação também podem ser classificados em operacional, conhecimento, gerencial e estratégico. Essa classificação baseia-se nos níveis hierárquicos existentes dentro da organização. No entanto, ela não é contrária a citada anteriormente, uma vez que ambas têm o mesmo objetivo: atender aos diversos níveis hierárquicos existentes em uma organização.

2.1.4. ERP (*Enterprise Resource Planning*)

Durante a década de 1970, os sistemas MRPs (*Material Resource Planning*) eram amplamente utilizados pelas indústrias para gerenciar a produção e, realizavam tarefas como "calcular a quantidade de itens requisitados em um dado momento com base nas necessidades de produtos finais, nas informações das estruturas de produto e nos dados de estoque" (SLACK *et al*, 1998 *apud* ZANCUL, 2000, p. 56).

A partir da década de 1980, novas funcionalidades foram adicionadas aos MRPs e, esses sistemas passaram a abranger tanto a gestão de materiais quanto a gestão de manufatura. Devido a isso, esses sistemas passaram a ser conhecidos como MRP II (*Manufacturing Resource Planning*). De acordo com Corrêa e Gianesi (1994 *apud* SOUZA, 2000, p.11),

> O princípio básico do MRP II é o princípio do cálculo de necessidades, uma técnica de gestão que permite o cálculo, viabilizado pelo uso de computador, das quantidades e dos momentos em que são necessários os recursos de manufatura (materiais, pessoas, equipamentos, entre outros), para que se cumpram os programas de entrega de produtos com um mínimo de formação de estoques.

Ao final da década de 1980 surgiu a necessidade de integrar as informações da gestão de materiais e manufatura a outras informações provenientes de outros departamentos da organização. Dessa forma, os sistemas de informação começaram a ser integrados, e passaram a atender não apenas a um departamento de uma organização, mas à organização como um todo. É nesse contexto que surgem os primeiros sistemas ERP (*Enterprise Resource Planning*).

Um sistema ERP representa um pacote comercial de software que oferece suporte as atividades realizadas em uma organização. São compostos por módulos que acessam um banco de dados central. Esse banco de dados possui as informações sobre as operações da organização.

De acordo com Deloitte Consulting (1998 *apud* SOUZA, 2000, p.11) um ERP representa "um pacote de software de negócios que permite a uma companhia automatizar e integrar a maioria de seus processos de negócio, compartilhar práticas e dados comuns através de toda a empresa e produzir e acessar informações em um ambiente de tempo real".

2.1.4.1. Características dos sistemas ERP

Uma das principais características dos sistemas ERP é a capacidade de centralizar informações e integrar os diversos departamentos de uma organização. Além disso, para Souza (2000) algumas das principais características de um sistema ERP são:

- **Pacotes comerciais de software**: são utilizados para resolver dois problemas que ocorrem durante o desenvolvimento de sistemas para empresas específicas: o não cumprimento de prazos e de orçamentos.
- **São desenvolvidos a partir de modelos-padrão de processos**: ERPs não são desenvolvidos para clientes específicos. O objetivo desses sistemas é atender a requisitos genéricos do maior número possível de organizações. Dessa forma, os ERPs incorporam modelos de negócio que são adquiridos através da experiência acumulada pelas empresas desenvolvedoras de sistemas.
- **São integrados**: um ERP oferece compartilhamento de informações comuns entre os módulos disponíveis e, a verificação cruzada de informações entre diferentes partes do sistema.
- **Têm grande abrangência funcional**: um sistema ERP abrange um grande número de funcionalidades, dessa forma, um único sistema é capaz de atender a toda a hierarquia organizacional.
- **Utilizam um banco de dados corporativo**: "entre as diversas formas de se desenvolver sistemas totalmente integrados está à utilização de um banco de dados centralizado" (SOUZA, 2000, p.16).
- **Requerem procedimentos de ajuste**: através da adaptação, um sistema ERP é preparado para ser utilizado por uma determinada empresa. De acordo com Souza

(2000, p.16), “a adaptação pode ser entendida como um processo de eliminação das discrepâncias, entre o pacote e a empresa”.

Além das características citadas anteriormente, Souza (2000) aponta alguns conceitos importantes relacionados aos ERPs:

- **Funcionalidades**: representam o conjunto de funções que estão embutidas em um sistema ERP. Segundo Souza (2000, p. 17), “o termo funcionalidade é utilizado para representar o conjunto total de diferentes situações que podem ser contempladas e diferentes processos que podem ser executados no sistema”.
- **Módulos**: representam os menores conjuntos de funções que podem ser adquiridos, independentemente, em um sistema ERP. Normalmente, os módulos representam as divisões departamentais de uma organização (vendas, financeiro, produção, etc). Souza (2000, p.17) afirma que os sistemas ERP são divididos em módulos para “possibilitar que uma empresa implemente apenas aquelas partes do sistema que sejam de seu interesse, e, mesmo que a empresa deseje implementar todo o sistema, possa fazê-lo em etapas para simplificar o processo”.
- **Parametrização**: representa o processo de configuração de um sistema ERP para uma determinada organização “através da definição dos valores de parâmetros já disponibilizados no próprio sistema. Parâmetros são variáveis internas ao sistema que determinam de acordo com o seu valor, o comportamento do sistema” (SOUZA, 2000, p.17).
- **Customização**: representa as modificações realizadas em um sistema ERP para que este se adapte à organização pela qual será utilizado. As customizações podem ser realizadas pela empresa desenvolvedora do sistema ERP (através da modificação do código-fonte do sistema). Ou, pela empresa cliente, através da construção de módulos adicionais que se comunicam que o sistema ERP.
- **Localização**: representa a adaptação (através de parametrizações ou customizações) de um sistema ERP desenvolvido em um determinado país para a utilização em outro. A localização leva em consideração aspectos como impostos, taxas, leis e procedimentos comerciais.
- **Atualização de versões**: é o processo através do qual novas funcionalidades, ou correções, são disponibilizadas para as organizações que utilizam determinado sistema ERP.

2.1.4.2. Arquitetura dos sistemas ERP

Sistemas ERP costumam ser divididos em módulos. Cada módulo representa um departamento de uma organização. Exemplos de módulos são: vendas, compras, produção, etc.

No centro de um sistema ERP fica localizado um banco de dados central. Esse banco é acessado pelos módulos do sistema ERP e contém todas as informações da organização De acordo com Zancul (2000, p. 60), utiliza-se o termo banco de dados central "no sentido da definição centralizada da lógica de armazenamento e manipulação de dados, uma vez que, fisicamente, os dados podem estar distribuídos em mais de uma base de dados."

A Figura 2 ilustra a arquitetura de um sistema ERP.

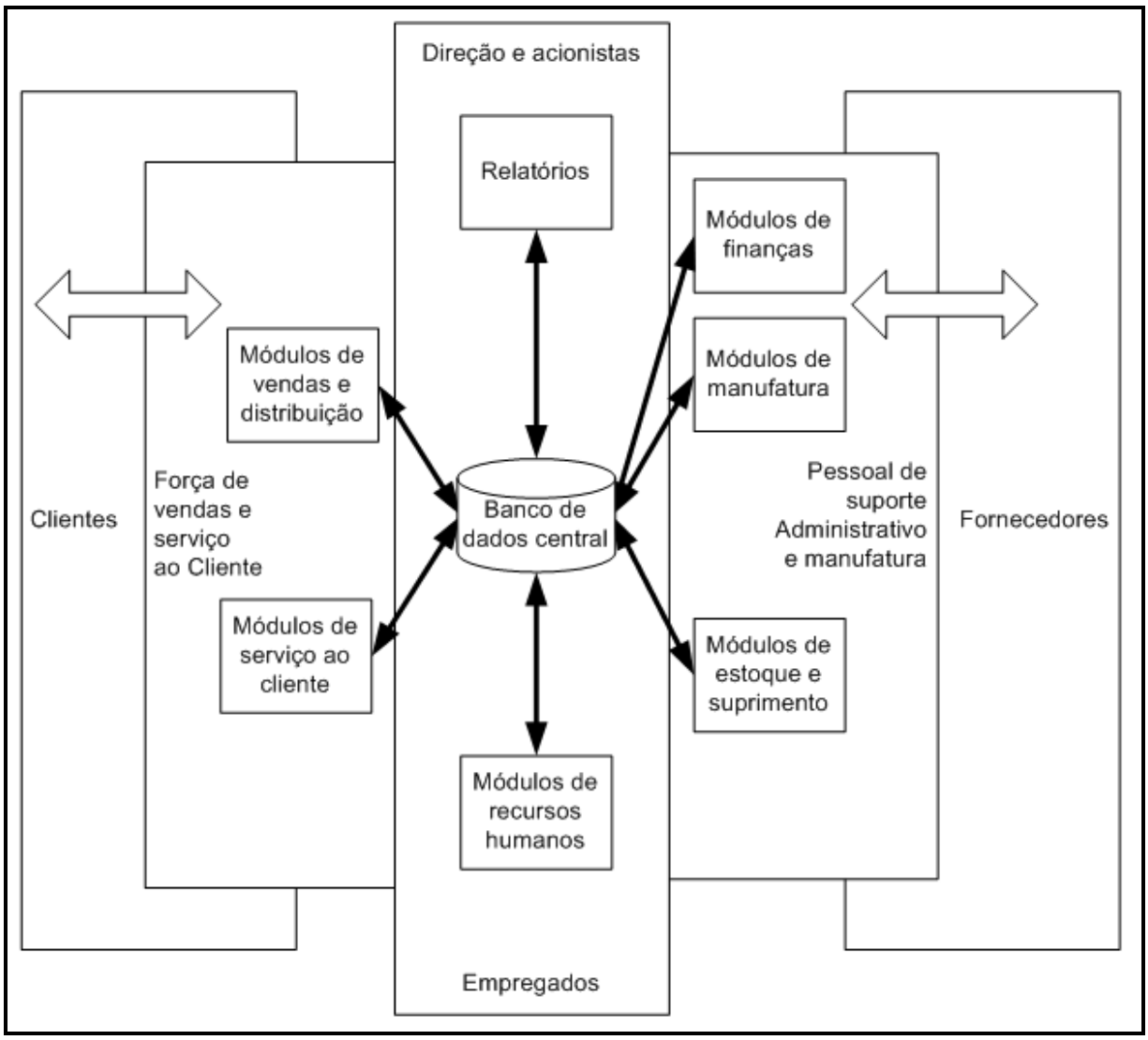

Figura 2 – Arquitetura de um sistema ERP
Fonte: Davenport (1998 *apud* SOUZA, 2000)

Os sistemas ERP atuais utilizam a arquitetura cliente/servidor. Nessa arquitetura, uma máquina cliente solicita serviços a um servidor. Normalmente o banco de dados central fica localizado no servidor, todo o processamento também é realizado no servidor.

De acordo com Souza (2000), a arquitetura cliente/servidor pode ser dividida em três tipos de processamento:

- Duas camadas (*two-tier*);
- Três camadas (*three-tier*);
- N camadas (*n-tier*).

A Figura 3 ilustra um sistema ERP em três camadas.

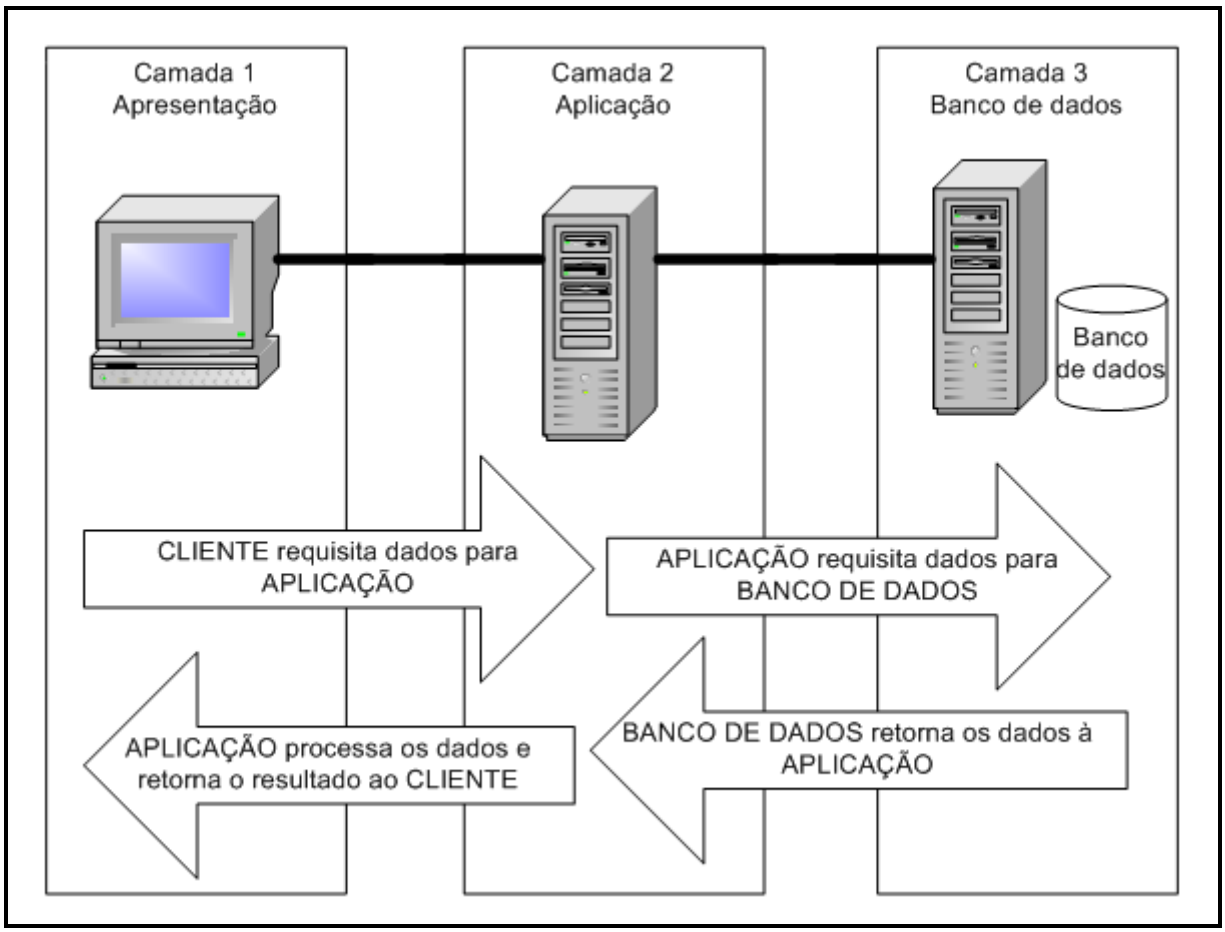

Figura 3 – Sistema ERP em três camadas
Fonte:Bancroft *et al*. (1998 *apud* SOUZA, 2000)

Souza (2000, p. 21) afirma que, no caso dos sistemas ERP,

> As aplicações podem ser divididas em três partes principais: a apresentação dos dados, os programas que processam as transações e o banco de dados. Estes três componentes podem estar localizados todos no mesmo computador (arquitetura mainframe tradicional), divididas em dois computadores na arquitetura cliente-servidor em duas camadas, com o computador servidor realizando o processamento

> do banco de dados e dos programas e o computador cliente realizando o processamento da apresentação, e finalmente, em uma arquitetura cliente-servidor de três camadas, o banco de dados pode ser processado em um servidor, chamado de servidor de banco de dados e os programas processados em um segundo servidor, chamado de servidor de aplicações e o cliente realizando a apresentação dos dados.

Os sistemas ERP atuais costumam utilizar a arquitetura cliente/servidor de três camadas devido ao alto grau de escalabilidade que essa abordagem oferece. Dessa forma, conforme a demanda de processamento cresce, é possível aumentar o número de servidores utilizados e, conseqüentemente, aumentar o poder de processamento.

2.1.5. Tecnologia da Informação

Os sistemas de informação baseados em computador utilizam a tecnologia da computação e as telecomunicações para realizar a entrada, o processamento, a saída e o armazenamento de dados.

A Tecnologia da Informação (TI) pode ser conceituada como um conjunto de recursos utilizados para coletar, armazenar, processar e distribuir informação. De acordo com Andrade, Audy e Cidral (2005), a TI também abrange os métodos, técnicas e ferramentas utilizadas para planejar, desenvolver e dar suporte aos processos de utilização da informação.

Segundo Andrade, Audy e Cidral (2005, p.114),

> Com o avanço da tecnologia da informação, os recursos de hardware e software passaram a ser componentes dos chamados sistemas de informação baseados em computador. O uso desse tipo de sistema de informação está pautado na melhoria da capacidade de processamento, na qualidade da informação oferecida e na relação custo/benefício proporcionadas pelo emprego das ferramentas oferecidas pela informática e pelas telecomunicações. Nesse sentido, o objetivo da tecnologia da informação é dotar os sistemas de informação de maior efetividade.

Atualmente as organizações têm acesso a uma quantidade de dados muito grande. Transformar esses dados em informação útil à organização, gastando pouco tempo e recursos, é uma tarefa difícil. A TI ajuda a tornar as atividades realizadas pelos sistemas de informação mais ágeis, eficientes e eficazes. A TI alcança esses objetivos através da informatização das rotinas realizadas por um sistema de informação. Como conseqüência, os sistemas de informação baseados em computador, têm se tornado cada vez mais uma ferramenta indispensável dentro das organizações. Isso ocorre porque os sistemas de informação podem ser empregados em todos os níveis hierárquicos existentes dentro da organização, auxiliando

tanto nas rotinas diárias executadas no nível operacional, quanto nas tarefas executada pelos gerentes.

Para O'Brien (2002, p.3), "sistemas e tecnologias da informação se tornaram um componente vital ao sucesso das empresas e organizações". Para esse autor, compreender os sistemas de informação é tão importante quanto compreender outras áreas como contabilidade, finanças, administração, entre outras.

2.2. Engenharia de software

Quando os primeiros softwares começaram a ser desenvolvidos, ninguém poderia ter imaginado o papel que este assumiria. Hoje o software é responsável pela criação e evolução de diversos tipos de tecnologias, como a engenharia genética e as telecomunicações. O software está presente no cotidiano de todas as pessoas, ele é utilizado desde tarefas rotineiras como acesso a Internet até o controle de usinas nucleares. Mas, o que ninguém poderia ter imaginado também, é que seriam necessários mais recursos para corrigir, adaptar e aperfeiçoar os softwares do que para criação de novos softwares.

Pressman (2006, p.12), afirma que:

> Hoje em dia o software assume um duplo papel. Ele é o produto e, ao mesmo tempo, o veículo para entrega do produto. Como produto ele disponibiliza o potencial de computação presente no hardware do computador ou, mais amplamente, por uma rede de computadores acessíveis pelo hardware local. Quer resida em um telefone celular, quer opere em um computador de grande porte, o software é um transformador de informação – produzindo, gerindo, adquirindo, modificando, exibindo ou transmitindo informações que podem ser tão simples como um único bit ou tão complexas quanto uma apresentação multimídia. Como veículo usado para entrega do produto, o software age como base para o controle de outros programas (ferramentas e ambientes de software).

Softwares que atendem a necessidade de seus usuários podem, efetivamente, modificar as coisas para melhor. Porém, um software que não apresenta desempenho satisfatório, não é fácil de manter e, principalmente, não é fácil de utilizar, provavelmente irá causar muitos transtornos.

Conforme a importância do software foi aumentando, foi necessário o desenvolvimento de tecnologias que pudessem facilitar o seu desenvolvimento, tornando esse processo mais rápido, fácil e menos trabalhoso. Algumas dessas tecnologias estão voltadas para domínios específicos, como o desenvolvimento de aplicações *web*, outras estão voltadas para domínios tecnológicos, como sistemas orientados a objetos.

A engenharia de software surge, neste contexto, para tentar aplicar princípios de engenharia ao desenvolvimento de software, com a finalidade de produzir software de alta qualidade e baixo custo.

Para Pedrycz e Peters (2001, p. V), a engenharia de software, cada vez mais, está sendo vista como a "aplicação dos métodos e tecnologias da engenharia para planejar, especificar, desenhar, implementar, validar, testar, medir, manter e aprimorar os sistemas de software".

Um conjunto de princípios deu origem à engenharia de software. Rezende (2005) cita alguns deles:

- Formalidade para evitar a dependência de pessoas ou processos;
- Abstração para identificar pontos importantes;
- Decomposição para subdividir problemas complexos;
- Generalização para difundir soluções e reutilizar resultados;
- Flexibilização para facilitar mudanças.

A indústria de software se tornou muito grande. O programador solitário de anos atrás foi substituído por equipes de especialistas em software, onde cada especialista se dedica a uma parte da tecnologia. No entanto, Pressman (2006), afirma que as questões feitas aos programadores de antigamente, continuam sendo as mesmas feitas para as equipes de especialistas em software dos dias atuais. Algumas dessas questões são:

1. Por que leva tanto tempo para concluir o software?
2. Por que os custos de desenvolvimento são tão altos?
3. Por que não é possível achar todos os erros antes de entregar o software aos clientes?
4. Por que se gasta tanto tempo e esforço mantendo os programas existentes?
5. Por que se continua a ter dificuldade em avaliar o progresso enquanto o software é desenvolvido e mantido?

A engenharia de software se preocupa em responder e, conseqüentemente, encontrar soluções viáveis para estas e outras questões relativas aos problemas encontrados para desenvolver software.

Sommerville (2003, p.4) define a engenharia de software da seguinte forma:

> A engenharia de software é uma disciplina da engenharia, cuja meta é o desenvolvimento de sistemas de software com boa relação custo-benefício. O software é abstrato e intangível, não é restrito por materiais nem regido por leis físicas ou por processos industriais. De certa maneira, isso simplifica a engenharia de software, por não haver limitações físicas para o potencial do software. Por outro lado, contudo, essa ausência de limitações naturais significa que o software pode

> facilmente se tornar bastante complexo, e conseqüentemente, muito difícil de ser compreendido.

É possível perceber que o software é mais do que uma série de documentos que resultam em linhas de código de um programa executável. O software é um produto que oferece funcionalidades, de acordo com as necessidades dos clientes que solicitam a construção do sistema de software.

2.2.1. Processo e ciclo de vida de software

Um processo de software é um conjunto de atividades com *feedback*, realizadas com a finalidade de produzir e manter os sistemas de software. Esse conjunto de atividades produz uma série de artefatos que resultam em um programa executável.

A visão tradicional do processo de software envolve uma seqüência linear de atividades. No entanto, esta visão está dando lugar a um processo onde ocorre uma sobreposição das atividades necessárias para especificar, projetar, testar e manter o software. O *feedback* dessas atividades ajuda a entender o que é necessário para criar um produto.

Segundo Sommerville (2003, p.36), um modelo de processo de software é "uma representação abstrata de um processo de software. Cada modelo de processo representa um processo a partir de uma perspectiva particular, de uma maneira que proporciona apenas informações parciais sobre o processo".

Existem diversos modelos de processo de software, porém, algumas atividades fundamentais são comuns a todos os modelos. Pressman (2006) propõe um conjunto de cinco atividades básicas (apresentadas na Tabela 2), enquanto Sommerville propõe um conjunto de quatro atividades (apresentadas na Tabela 3).

Tabela 2 – Atividades básicas do processo de software (Pressman)

Atividade	**Descrição**
Comunicação	Envolve a comunicação e colaboração entre os envolvidos com o projeto. Abrange, entre outras atividades, o levantamento de requisitos.
Planejamento	Descreve as atividades e riscos envolvidos, os recursos necessários, os produtos a serem produzidos e um cronograma de atividades.

Modelagem	Nesta atividade são criados modelos que permitem a clientes e desenvolvedores entender melhor os requisitos do software e o projeto que atenderá a estes requisitos.
Construção	Nesta atividade é gerado o código-fonte do software e os testes necessários para encontrar erros no código-fonte gerado.
Implantação	Nesta atividade o software é entregue ao cliente que avalia o produto que recebeu e fornece um *feedback.*

Fonte: Pressman (2006)

Tabela 3 – Atividades básicas do processo de software (Sommerville)

Atividade	**Descrição**
Especificação	Define as funcionalidades e restrições do software
Projeto e implementação	Produz o software de acordo com a especificação do mesmo.
Validação	Avalia o software para ter certeza de que o cliente receberá um produto que satisfaz as suas necessidades.
Evolução	Evolução natural do software para atender as necessidades do cliente. Pode ocorrer tanto através da inclusão ou manutenção de funcionalidades.

Fonte: Somerville (2003)

O processo de ciclo de vida de software "é o período de tempo que se inicia com um conceito para um produto de software e acaba sempre que o software deixa de estar disponível para utilização" (PEDRYCZ e PETERS, 2001, p. 36). Modelos de ciclo de vida de software representam as atividades e artefatos produzidos por estas atividades, e as interações que ocorrem durante o ciclo de vida.

2.2.1.1. Modelo em cascata (ou ciclo de vida clássico)

O modelo em cascata, ilustrado na Figura 4, é o mais antigo de todos os modelos de ciclo de vida. Muitos dos outros modelos derivaram deste. As atividades do modelo em cascata ocorrem de forma sistemática e seqüencial, iniciando com a identificação dos requisitos do sistema e finalizando com a manutenção do software.

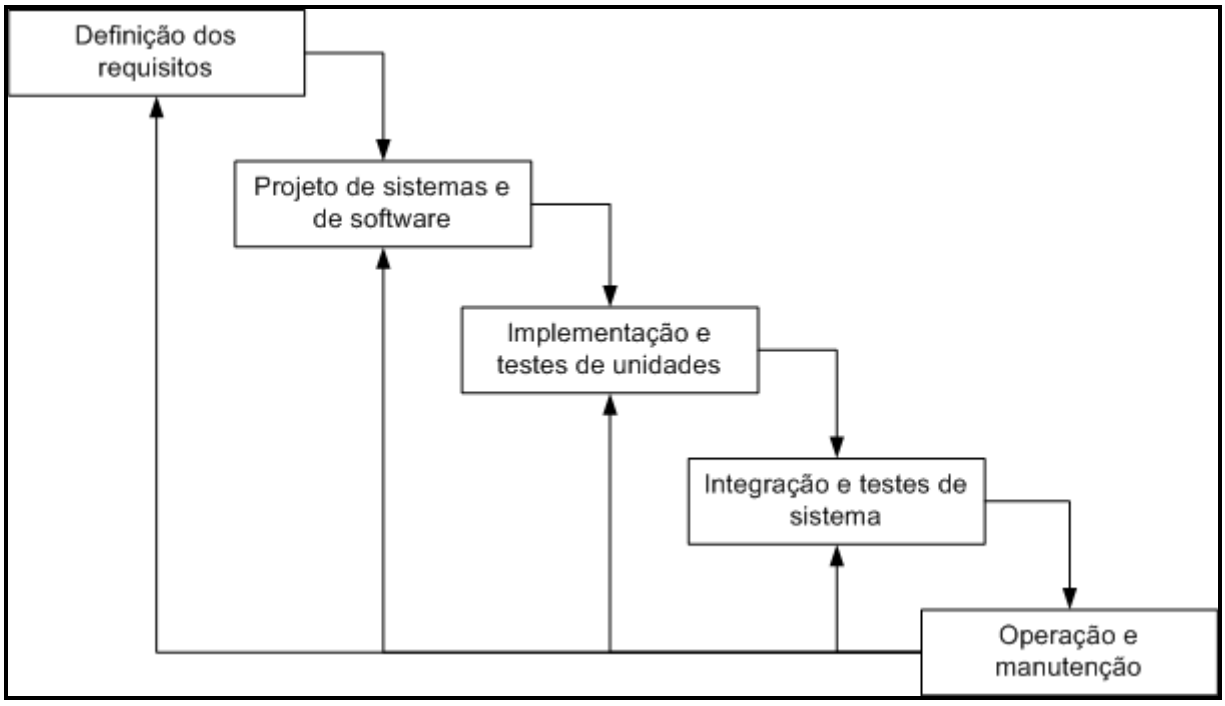

Figura 4 – Modelo em cascata
Fonte: Sommerville (2003)

As principais etapas desse modelo descrevem as atividades fundamentais do processo de desenvolvimento de software. De acordo com Sommerville (2003), as atividades desse modelo são:

- **Planejamento**: atividade que define como serão executadas todas as outras atividades do processo de desenvolvimento de software;
- **Levantamento de requisitos**: são identificados os requisitos que o sistema de software deve atender;
- **Análise de requisitos**: analisam-se os requisitos identificados com o intuito de ter certeza de que o software atenderá as necessidades dos usuários;
- **Projeto (*design*)**: desenvolvem-se modelos de componentes e seus conectores, ou seja, é definida a estrutura que o software deverá ter;
- **Implementação**: os requisitos são transformados em linhas de código gerando um produto executável;
- **Testes**: os artefatos gerados durante a implementação são testados para ter certeza de que atendem aos requisitos funcionais e não-funcionais;
- **Implantação**: o software é entregue ao cliente;
- **Manutenção**: a manutenção pode ocorrer de formas diferentes: através da correção de erros identificados pelos usuários, através da inclusão ou aprimoramento de

funcionalidades e, através da adequação do software a algum tipo de norma ou legislação.

Cada fase do modelo em cascata só se inicial quando a fase anterior for concluída e validada. O resultado de cada etapa deste modelo fornece um *feedback* aos envolvidos. Esse *feedback* pode ser utilizado para identificar problemas no decorrer do projeto. Sommerville (2003, p.38) afirma que:

> Devido aos custos de produção e aprovação de documentos, as iterações são onerosas e envolvem um 'retrabalho' significativo. Portanto, depois de um pequeno número de iterações, é normal suspender partes do desenvolvimento, como a especificação, e continuar com os estágios posteriores de desenvolvimento. Os problemas são deixados para solução posterior, são ignorados ou programados para serem solucionados. Essa suspensão prematura da definição de requisitos pode significar que o sistema não fará o que usuário quiser. Pode também levar a sistemas mal-estruturados, uma vez que os problemas de projeto são resolvidos por 'gambiarras' na implementação.

Apesar de ser o paradigma mais antigo da engenharia de software, Pressman (2006, p. 39) afirma que "nas duas últimas décadas, a crítica a esse modelo de processo tem provocado, mesmo em seus mais ardentes adeptos, questionamentos sobre sua eficácia". Alguns dos problemas encontrados neste modelo são os seguintes:

- Em projetos reais, dificilmente é seguido o fluxo seqüencial que o modelo propõe;
- Dificilmente todos os requisitos do software são identificados logo no início do projeto;
- Versões executáveis do projeto só estarão disponíveis no final do projeto.

2.2.1.2. Modelo de desenvolvimento evolucionário

No modelo de desenvolvimento evolucionário, uma implementação inicial, é desenvolvida e apresentada ao cliente. Através da avaliação do cliente são feitos aprimoramentos no software até que este resulte em um sistema adequado as necessidades do cliente. As atividades de especificação, desenvolvimento e validação são realizadas concorrentemente, com um rápido *feedback* entre as atividades.

Segundo Sommerville (2003, p.39), essa abordagem é mais eficaz do que o modelo em cascata, "no sentido de produzir sistemas que atendem às necessidades imediatas dos clientes". Uma vantagem visível neste modelo é que a especificação de requisitos pode ser feita gradativamente. No entanto, na perspectiva de engenharia e gerenciamento, Sommerville (2003) identifica três problemas:

1. **O processo não é visível**: não é possível produzir documentos que reflitam cada versão do sistema.
2. **Os sistemas freqüentemente são mal-estruturados**: as mudanças constantes tornam difícil modificar os sistemas.
3. **Podem ser exigidas ferramentas e técnicas especiais**: podem permitir desenvolvimento rápido, mas também podem ser incompatíveis com outras ferramentas e poucas pessoas podem ter o conhecimento necessário para utilizar estas ferramentas.

O modelo de desenvolvimento evolucionário pode ser uma boa opção para sistemas pequenos. Porém, em sistemas de grande porte, os problemas desse tipo de abordagem podem se tornar ainda maiores, à medida que o sistema for se tornando cada vez maior e mais complexo.

2.2.1.3. Modelo de prototipagem

A prototipagem, ilustrada na Figura 5, pode ser utilizada como um modelo independente, no entanto, é mais comum que seja utilizada como uma técnica que pode ser implementada em qualquer modelo de processo. O objetivo desse modelo é ajudar a todos os envolvidos no processo de desenvolvimento a entender melhor o que deve ser construído quando os requisitos estiverem confusos. De acordo com Pressman (2006), o protótipo serve como um "mecanismo para identificação dos requisitos do software".

A prototipação se inicia com a identificação dos objetivos gerais do software, através da identificação das necessidades conhecidas, a partir daí se definem as áreas que precisam de mais detalhes. As iterações ocorrem no formato de projetos rápidos. Segundo Pressman (2006, p. 42), "o projeto rápido concentra-se na representação daqueles aspectos do software que estarão visíveis para o cliente/usuário". O protótipo criado através do projeto rápido é implantado e o cliente pode avaliá-lo. O *feedback* do cliente será utilizado para refinar os requisitos do software.

Pressman (2006, p.43) afirma que:

> Apesar de problemas poderem ocorre, a prototipagem pode ser um paradigma efetivo para a engenharia de software. O importante é definir as regras do jogo no início; isto é, cliente e desenvolvedores devem estar de acordo que o protótipo é construído para servir como mecanismo de definição dos requisitos. Depois ele será descartado (pelo menos em parte), e o software real será submetido à engenharia com um olho na qualidade.

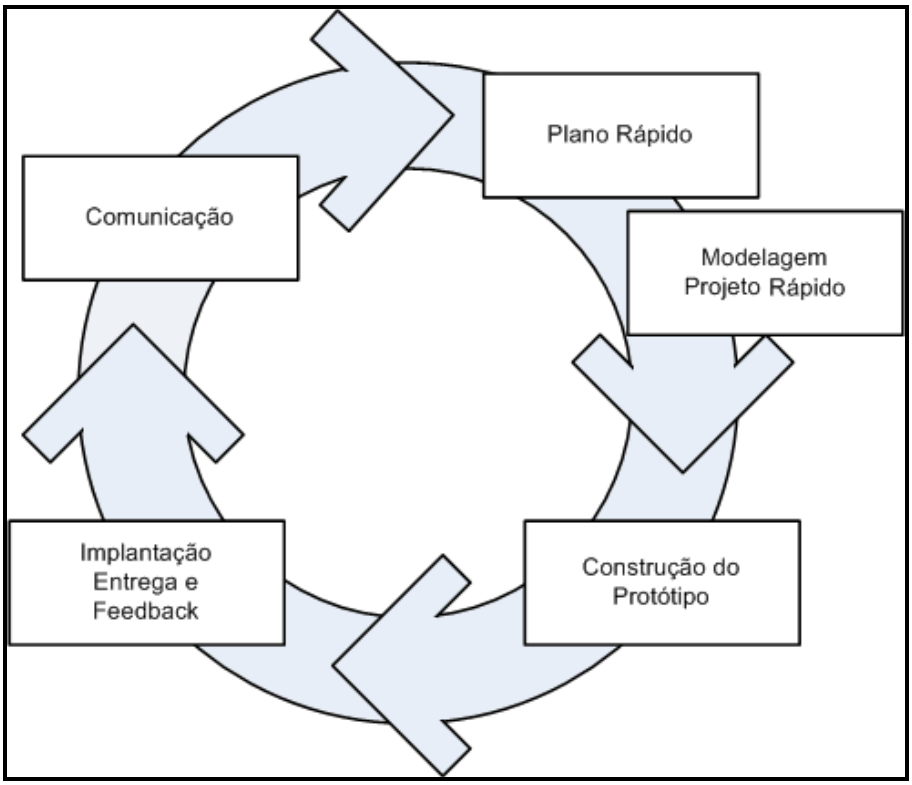

Figura 5 – Modelo de prototipagem
Fonte: Pressman (2006)

Uma vez que os clientes nem sempre sabem quais são as reais necessidades do sistema que será construído, definir todos os requisitos do software no início do processo de desenvolvimento é uma tarefa difícil. Assim, a prototipagem pode auxiliar a definir tais requisitos.

O ponto chave deste modelo é o desenvolvimento de protótipos com funcionalidades limitadas. Com base nas funcionalidades desenvolvidas, cliente e desenvolvedores, podem se certificar de que é isso que o sistema deve oferecer, antes de se comprometerem com um sistema final.

2.2.1.4. Modelo em espiral

O modelo em espiral, conforme ilustrado na Figura 6, combina características do modelo em cascata e de prototipagem. Oferece a iteratividade do modelo de protótipo com o controle oferecido pelo modelo em cascata.

Esse modelo não representa o processo de software como uma seqüência de atividades que fornecem um *feedback*, e sim como uma espiral. Dessa forma,

> Cada *loop* na espiral representa uma fase do processo de software. Assim, o *loop* mais interno pode estar relacionado à viabilidade do sistema, o *loop* seguinte, à definição de requisitos do sistema; o próximo, ao projeto do sistema, e assim por diante (SOMMERVILLE, 2003, p.45).

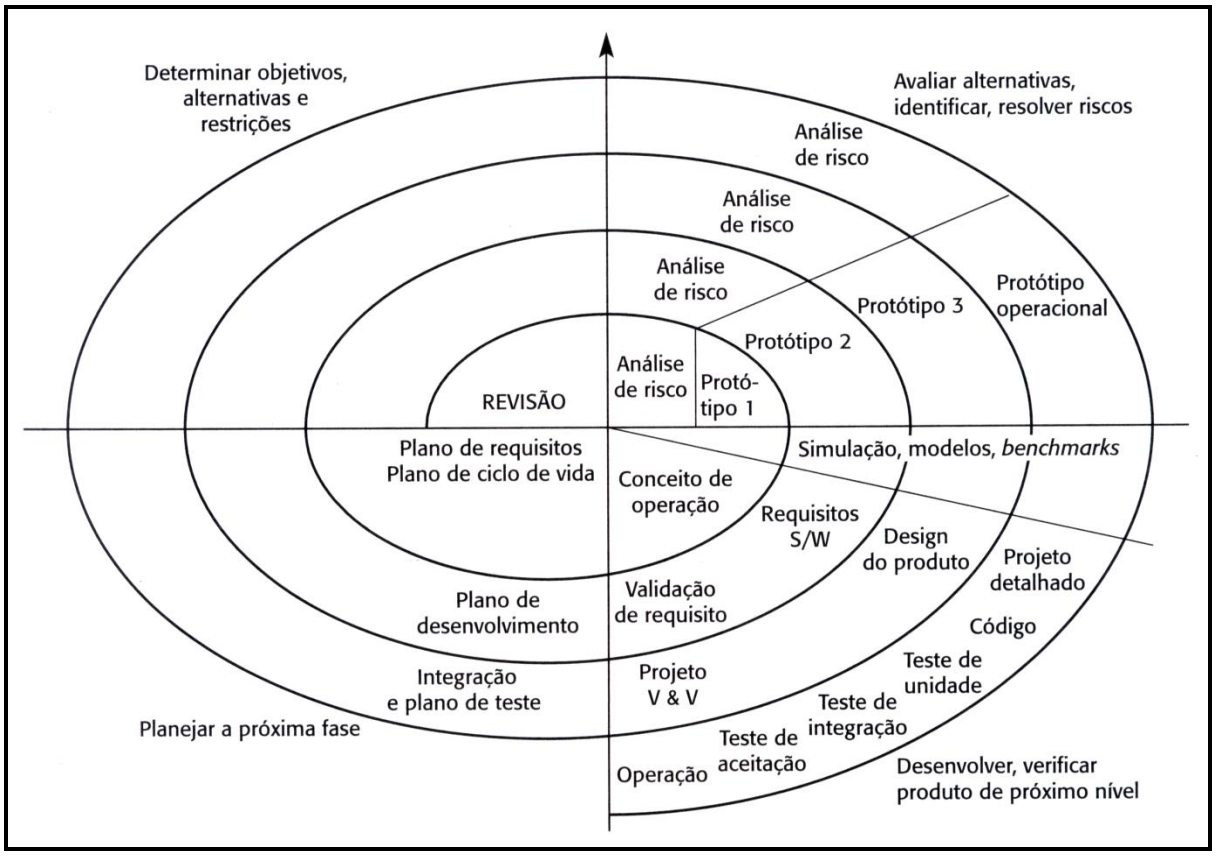

Figura 6 – Modelo em espiral
Fonte: Sommerville (2003)

De acordo com Sommerville (2003), cada *loop* da espiral é dividido em quatro atividades:

1. **Definição de objetivos**: definem-se os objetivos específicos para a fase atual, as restrições do processo, os riscos do projeto e, se prepara um plano de gerenciamento;
2. **Avaliação e redução de riscos**: para cada risco identificado, realiza-se uma análise para determinar quais medidas devem ser tomadas para evitar ou reduzir esses riscos;
3. **Desenvolvimento e validação**: após a avaliação dos riscos, escolhe-se um modelo de desenvolvimento adequado ao sistema;
4. **Planejamento**: o projeto é revisado e se toma a decisão de continuar ou não com o próximo *loop*. Se a decisão for sim, então são definidos os planos para as próximas fases.

O modelo em espiral aborda o desenvolvimento de software de uma forma realista, pois, à medida que o processo avança, clientes e desenvolvedores entendem melhor o que

precisa ser feito e os riscos associados a cada ciclo. No entanto, utilizar este modelo de processo de software, exige muita competência para identificação dos riscos e depende dessa competência para alcançar o sucesso.

2.2.1.5. Modelos de métodos formais

Os métodos formais permitem desenvolver software de uma forma consistente e não ambígua, através da utilização de um conjunto de atividades que levam a especificação matemática.

Os métodos formais eliminam uma variedade de problemas encontrados em outras abordagens da engenharia de software. Segundo Pressman (2006, p. 49) "ambigüidade, inconclusão e inconsistência podem ser descobertas e corrigidas facilmente – não por meio de revisões *ad hoc*, mas por meio da aplicação de análise matemática".

Apesar deste tipo de abordagem prometer softwares livres de defeitos, Pressman (2006), afirma que existem alguns pontos negativos que devem ser considerados:

- Atualmente este método é muito lento e trabalhoso;
- Existem poucos desenvolvedores preparados para aplicar métodos formais ao desenvolvimento de software, o que torna necessário um treinamento específico;
- A linguagem utilizada por métodos formais é de difícil entendimento para pessoas despreparadas tecnicamente.

Uma abordagem derivada dos métodos formais é conhecida como engenharia de software sala limpa. Este método "enfatiza a verificação matemática da correção antes que comece a construção do programa e a certificação da confiabilidade do software como parte da atividade de teste" (PRESSMAN, 2006, p.647).

A maior vantagem da abordagem da engenharia de software sala limpa é que este procura evitar o retrabalho exigido pela correção de erros.

2.2.1.6. Modelo incremental

O modelo incremental, ilustrado na Figura 7, aplica as etapas do modelo em cascata de maneira iterativa. Esse modelo aplica uma seqüência linear de forma racional com o passar do tempo. Cada seqüência produz incrementos que podem ser entregues.

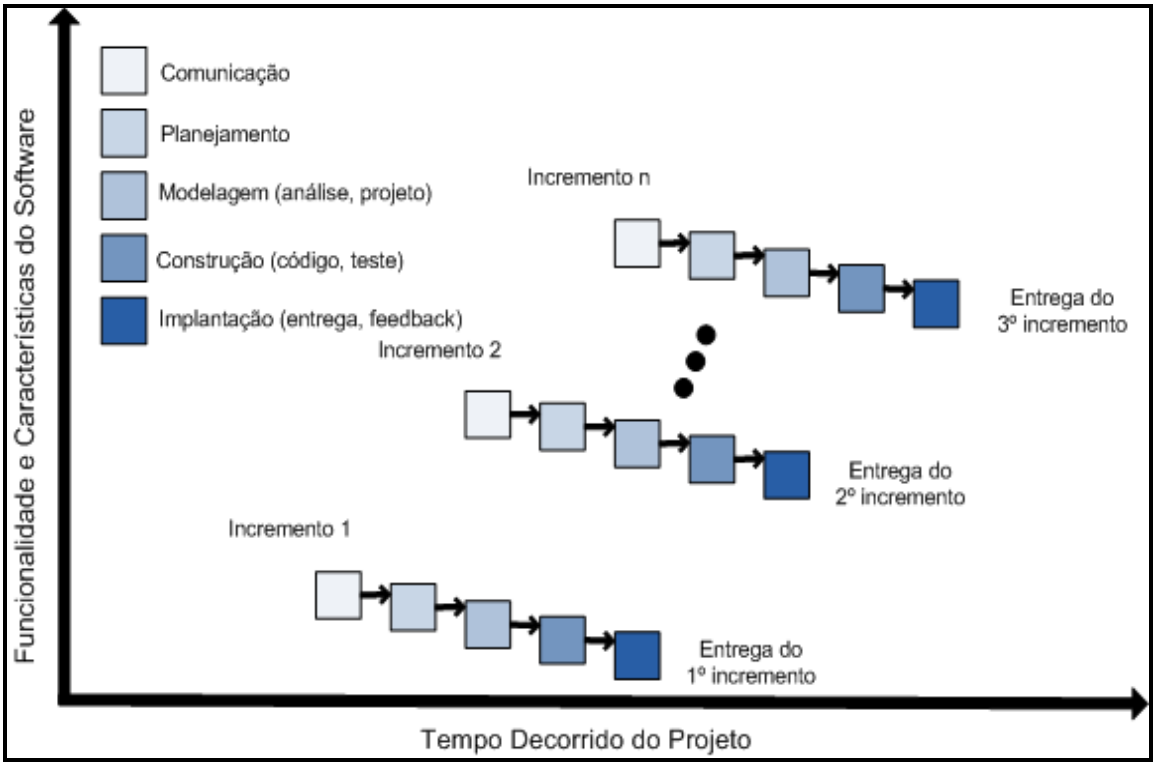

Figura 7 – Modelo incremental
Fonte: Pressman (2003)

O primeiro incremento deste modelo é chamado de núcleo do produto. Ou seja, “os requisitos básicos são satisfeitos, mas muitas características suplementares (algumas conhecidas, outras desconhecidas) deixam de ser elaboradas” (PRESSMAN, 2006, p.40).

O núcleo de produtos é entregue ao cliente para ser avaliado. Através do *feedback* fornecido pelo cliente, se elabora um plano para o próximo incremento. Este plano visa à modificação do produto para melhorar, corrigir ou inserir características e funcionalidades no produto.

O objetivo principal do modelo incremental é oferecer um produto operacional a cada incremento. Esse modelo é recomendado quando não existe mão de obra disponível para a implementação completa do produto.

2.2.1.7. Modelo RAD (*Rapid Application Development*)

O modelo RAD (*Rapid Application Development*), ilustrado na Figura 8, é um modelo incremental que enfatiza um ciclo de desenvolvimento curto. Este modelo pode ser considerado como uma adaptação do modelo em cascata, onde o desenvolvimento rápido é alcançado através do uso de uma abordagem de construção baseada em componentes.

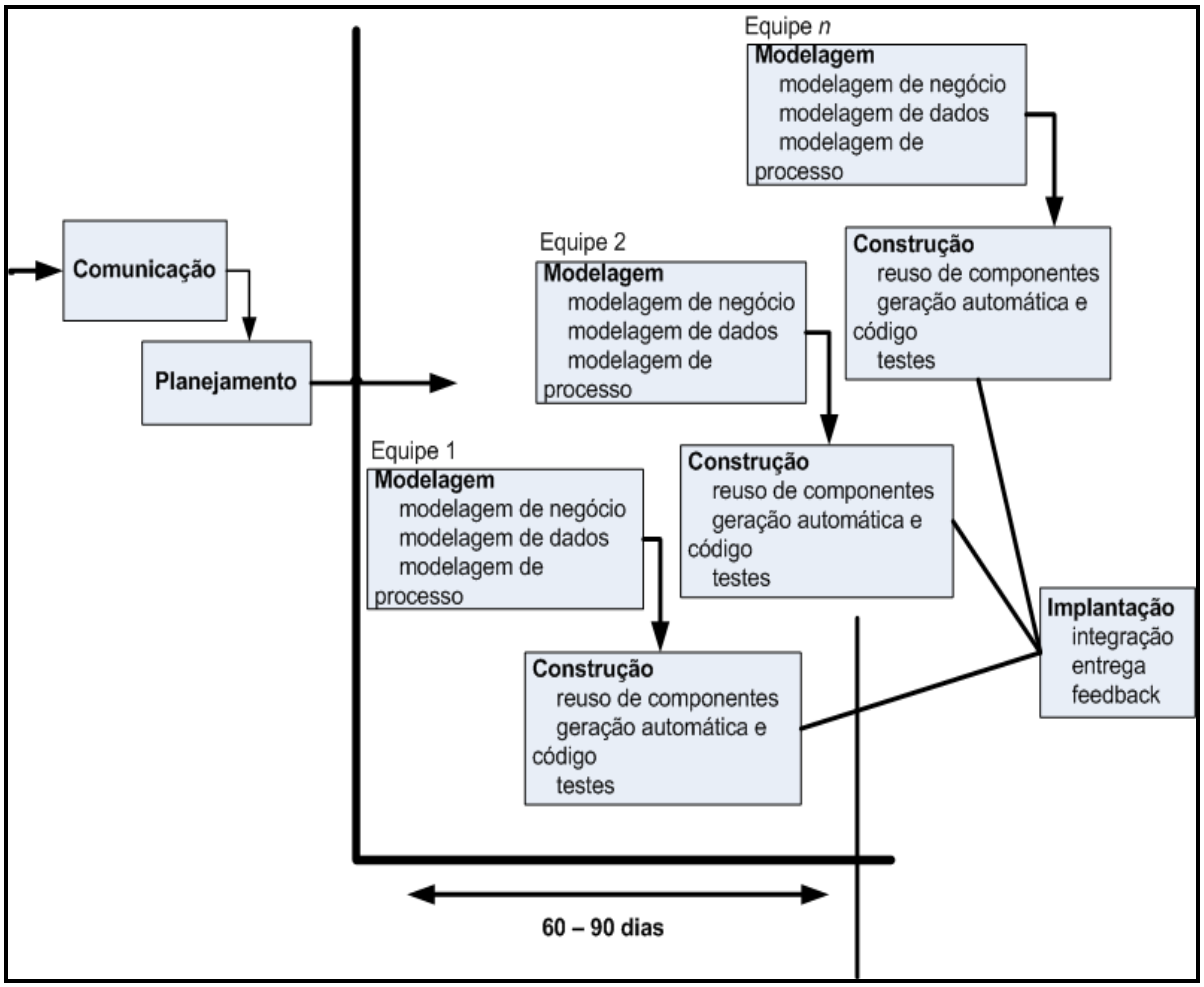

Figura 8 – Modelo RAD
Fonte: Pressman (2006)

Pressman (2006, p.41), descreve as atividades deste modelo da seguinte forma:

> A comunicação trabalha para entender os problemas de negócio e as características da informação que o software precisa acomodar. O planejamento é essencial, porque várias equipes de software trabalham em paralelo em diferentes funções do sistema. A modelagem abrange três das principais fases – modelagem de negócio, modelagem dos dados e modelagem dos processos – e estabelece representações de projeto que servem como base para as atividades de construção do RAD. A construção enfatiza o uso de componentes de software preexistentes e a aplicação da geração automática de código. Finalmente, a implementação estabelece a base para iterações subseqüentes, se necessárias.

Algumas desvantagens apontadas por Pressman (2006) nesse modelo são:

- Em projetos de grande porte, o RAD exige recursos humanos suficiente para gerar várias equipes;
- É necessário que todos os envolvidos no projeto tenham um nível de comprometimento muito grande com as atividades, caso contrário o modelo falhará;

- O sistema a ser desenvolvido deve ser adequadamente modularizado para suportar o desenvolvimento de componentes necessários;
- Se o sistema precisar de alto desempenho e for necessário adaptar as interfaces dos componentes, este modelo pode não ser adequado;
- Este modelo não é adequado quando os riscos técnicos forem altos.

2.2.2. Engenharia de requisitos

Entender os requisitos de um sistema está entre as mais difíceis atividades do desenvolvimento de software. Isso ocorre, principalmente, porque nem sempre os clientes sabem quais são reais necessidades do sistema a ser construído, ou, quando sabem, não conseguem expressar isso. Mesmo que os clientes saibam exatamente o que querem e consigam transmitir isso aos desenvolvedores, normalmente essas necessidades se modificam com o tempo.

A engenharia de requisitos visa o entendimento dos problemas, a determinação de soluções e a especificação de soluções testáveis, compreensíveis, manuteníveis e que atendam aos requisitos de qualidade do sistema.

Pressman (2006, p. 102) descreve os requisitos de software da seguinte forma:

> Um requisito de software é uma descrição dos principais recursos de um produto de software, seu fluxo de informações, comportamento e atributos. Em suma, um requisito de software fornece uma estrutura básica para o desenvolvimento de um produto de software. O grau de compreensibilidade, precisão e rigor da descrição fornecida por um documento de requisitos de software tende a ser diretamente proporcional ao grau de qualidade do produto resultante.

A engenharia de requisitos tem como foco principal, a definição e descrição do que o sistema de software deverá fazer para atender aos requisitos relacionados a ele. Segundo Pressman (2006), o processo de engenharia de requisitos é realizado através de etapas distintas que podem ocorrer em paralelo, sendo adaptadas as necessidades do projeto:

- **Concepção**: depois de identificada uma necessidade, tenta-se entender o problema, compreender quais são os envolvidos com a solução, determinar um mecanismo de comunicação e colaboração entre usuários e desenvolvedores e identificar a natureza da solução.
- **Levantamento**: tenta-se identificar os objetivos do sistema, entender como o sistema será utilizado pelo cliente, ou seja, identificar as necessidades iniciais do cliente. Essa é uma atividade difícil, pois, os requisitos estão em constante mudança.

- **Elaboração**: nesta atividade, as informações obtidas durante a concepção e os requisitos identificados durante o levantamento são refinados. A partir daí são desenvolvidos modelos técnicos das funcionalidades, características e restrições do sistema. Pressman (2006, p. 119) descreve essa fase como "uma ação de modelagem de análise composta por várias tarefas de modelagem e refinamento".
- **Negociação**: os envolvidos com o projeto ordenam os requisitos propostos e iniciam um processo de avaliação sobre os riscos envolvidos em cada requisito. Inicia-se uma negociação para eliminar ou modificar os requisitos iniciais.
- **Especificação**: pode ter significados diferentes para pessoas diferentes. A especificação pode incluir desde documentos escritos até modelos matemáticos formais. De modo geral, a especificação serve como fundamento para as etapas que virão depois delas. É um modelo de funcionalidades, desempenho e restrições do sistema.
- **Validação**: "examina a especificação para garantir que todos os requisitos do software tenham sido declarados de modo não ambíguo; que as inconsistências, omissões e erros tenham sido detectados e corrigidos" (PRESSMAN, 2006, p.120). Ou seja, procura garantir que o sistema atende as normas determinadas para o projeto.
- **Gestão**: inicia-se com a identificação dos requisitos e estende-se por todo o processo de engenharia de requisitos. Depois de identificados os requisitos, são desenvolvidas tabelas de rastreamento. Essas tabelas relacionam os requisitos com outros aspectos do sistema.

Nem sempre é possível que clientes e desenvolvedores trabalhem em conjunto durante a engenharia de requisitos. Às vezes os clientes possuem opiniões conflitantes, ou estão em outras cidades ou países. Em casos como esse, existe uma série de passos que podem ser seguidos para iniciar a engenharia de requisitos. Pressman (2006) identifica alguns desses passos:

- **Identificação dos interessados**: são identificadas todas as pessoas que podem se beneficiar com o desenvolvimento do sistema.
- **Reconhecimento de diversos pontos de vista**: como pode existir uma grande diversidade de pessoas interessadas no sistema, é necessário identificar o ponto de vista de cada um desses interessados para que o sistema possa, efetivamente, atender a todos.
- **Trabalho em busca da colaboração**: devem ser identificados os requisitos com os quais todos os interessados concordam e, os requisitos que entram em conflito com

outros requisitos. Os interessados podem colaborar com o corte de requisitos, mas também é possível que um gerente de negócio decida quais são os requisitos que passam pelo corte (PRESSMAN, 2006).

- **Formulação das primeiras questões**: são formuladas questões que devem ser respondidas pelos interessados no sistema, visando melhorar o entendimento do problema entre os envolvidos.

2.2.3. Qualidade de software

Garantir a qualidade de software é uma tarefa difícil. Além de atender aos seus requisitos e, portanto, satisfazer a necessidade de seus usuários, um software de qualidade deve possuir uma série de características, tanto do ponto de vista dos usuários quanto dos desenvolvedores. Para os usuários de um sistema, é importante que este tenha usabilidade, desempenho, segurança, etc. Já os desenvolvedores, além de garantir que os softwares tenham as características que os usuários desejam, têm que garantir que o software será flexível e manutenível.

Pressman (2006, p. 580) define qualidade de software como sendo a "conformidade com requisitos funcionais e de desempenho explicitamente declarados, normas de desenvolvimento explicitamente documentadas e características implícitas, que são esperadas em todo software desenvolvido profissionalmente".

A garantia de qualidade (*Quality Assurance* – QA) é composta por um conjunto de atividades que definem a estrutura necessária para garantir a qualidade de software. O processo de QA, também, define padrões que devem ser aplicados ao processo de desenvolvimento de software. Sommerville (2003) define dois desses padrões:

- **Padrões de produto**: são aplicados aos produtos durante o processo de desenvolvimento. Incluem padrões de documentos de requisitos, padrões para documentação do sistema e padrões de codificação.
- **Padrões de processo**: definem os processos a serem seguidos durante o desenvolvimento. Incluem definições de especificação, processos de projeto e validação, descrição dos documentos que devem ser gerados durante o processo, etc.

Segundo Sommerville (2003, p. 461), os padrões de produto e de processo estão diretamente relacionados, pois, "em muitos casos, os padrões de processo incluem atividades específicas de processo que asseguram que os padrões de produto sejam seguidos".

Existem diversos motivos pelos quais os padrões de software são importantes. Sommerville (2003) cita três deles:

1. Padrões fornecem um encapsulamento das melhores práticas, ou pelo menos, das mais adequadas;
2. Disponibilizam uma infra-estrutura através da qual o processo de garantia de qualidade pode ser implementado;
3. Os padrões ajudam a diminuir o esforço de aprendizado, pois, garante que todos dentro de uma organização utilizam as mesmas práticas.

O planejamento da qualidade de software deve se iniciar nos estágios iniciais do processo de software. Um plano de qualidade deve definir quais são as qualidades desejadas ao sistema e como essas qualidades serão avaliadas.

Segundo Humphrey (1989 *apud* SOMMERVILLE, 2003), um plano de qualidade deve incluir:

1. **Introdução sobre o produto**: uma breve descrição sobre o produto e o mercado pretendido, bem como, as expectativas relacionadas à qualidade;
2. **Planos para o produto**: datas de liberação do produto, planos de distribuição e prestação de serviços relacionados ao produto;
3. **Descrição de processo**: processos de desenvolvimento e gerenciamento que serão utilizados para o produto;
4. **Metas de qualidade**: identificação e justificativa dos atributos de qualidade do produto;
5. **Riscos e gerenciamento de riscos**: identificação dos riscos que podem afetar a qualidade do produto e quais medidas devem ser tomadas para evitar tais riscos.

O controle de qualidade supervisiona o processo de desenvolvimento de software com a finalidade de garantir que os procedimentos e padrões de qualidade estão sendo seguidos. Uma das abordagens para controle de qualidade é a revisão de qualidade (ou revisões de software). Segundo Pressman (2006, p. 582),

> As revisões de software são como um "filtro" para o processo de engenharia de software. Isto é, as revisões são aplicadas em vários pontos durante a engenharia de software e servem para descobrir erros e defeitos que podem depois ser removidos. As revisões de software "purificam" as atividades da engenharia de software.

As revisões devem ser executadas por um grupo de pessoas que examinarão parte ou todo o processo de software. As conclusões dessas revisões são encaminhadas as pessoas responsáveis pelas correções necessárias.

Embora muitas pessoas afirmem que não é possível medir um software, existe uma série de medidas e métricas que fornecem valores para este tipo de avaliação. As medidas de software são utilizadas para obter valores numéricos que quantifiquem atributos de qualidade. Através desses números é possível avaliar a qualidade dos produtos ou processos submetidos à engenharia de software.

A medida de software é definida por McClure (1994 *apud* PETERS e PEDRYCZ, 2001, p.254) como "mapeamento de um conjunto de objetos no mundo da engenharia de software em um conjunto de construções matemáticas, tais como números ou vetores de números"

As métricas de software estão focalizadas em atributos específicos da engenharia de software e são obtidas durante as tarefas de análise, projeto, codificação e testes. Segundo Pressman (2006), as palavras medida, medição e métrica são utilizadas intercambiadamente. Porém, é necessário fazer uma distinção entre seus significados. De acordo com Pressman (2006):

- **Medida**: é a indicação quantitativa da extensão, quantidade, dimensão, capacidade ou tamanho de algum atributo de um produto ou processo;
- **Medição**: é o ato de determinar uma medida;
- **Métrica**: medida quantitativa do grau em que um software possui algum atributo.

As métricas de software são úteis apenas quando são validadas de forma que seja possível provar seu valor. Para isso, Pressman (2006), propõe algumas validações:

- As métricas devem possuir propriedades matemáticas desejáveis, ou seja, os valores obtidos através das métricas devem estar em um intervalo, como por exemplo, entre zero e um, onde zero representa ausência total e um representa o valor máximo.
- Se uma métrica representar uma característica de software que pode aumentar ou diminuir, os valores da métrica também devem aumentar ou diminuir na mesma proporção;
- Cada métrica deve medir um fator independente de outros fatores, ela deve se adequar ao sistema e funcionar em qualquer linguagem de programação ou domínio de sistema.

Além de quantificar as propriedades do software, as métricas do software auxiliam os engenheiros de software a medir os processos que envolvem o software, os recursos necessários a um projeto e os artefatos relevantes ao esforço de desenvolvimento.

2.2.4. Testes de software

Durante o processo de engenharia de software, procura-se garantir que o sistema que está sendo desenvolvido atenda aos mais altos padrões de qualidade. Para isso, são empregadas as melhores técnicas de desenvolvimento e modelagem, visando sempre gerar um produto final sem falhas.

Pressman (2006, p.290), afirma que

> O teste oferece efetivamente o último reduto no qual a qualidade pode ser avaliada e, mais pragmaticamente, erros podem ser descobertos. Mas o teste não deve ser encarado como rede de proteção. Como sabiamente se diz, "Você não pode testar a qualidade. Se ela não estiver lá antes de você começar a testar, ela não estará lá quando terminar de testar". A qualidade é incorporada ao software durante o processo de engenharia de software. A aplicação adequada de métodos e ferramentas, revisões técnicas formais efetivas e gerência e medição sólidas, todas levam à qualidade que é confirmada durante o teste (PRESSMAN, 2006, p. 290).

Testes de software são conjuntos de atividades realizadas com a finalidade de detectar falhas em um sistema. Um bom teste tem alta probabilidade de encontrar algum tipo de falha, não é redundante e não deve ser nem muito simples nem muito complexo.

De acordo com Dias (2008), quando se trata de testes de software, é necessário explicitar as diferenças existentes entre defeitos, erros e falhas. Essa diferenciação é feita por esse autor utilizando os conceitos propostos pelo IEEE (*Institute of Electrical and Electronics Engineers*), da seguinte forma:

- **Defeito**: representa um ato inconsistente cometido por um indivíduo ao tentar resolver um problema ou utilizar uma ferramenta ou tecnologia.
- **Erro**: representa, por exemplo, a diferença entre o valor obtido e o valor esperado, ou seja, é um resultado inesperado e, normalmente, incorreto na execução de um programa.
- **Falha**: podem ser causadas por erros, representam um comportamento operacional diferente do esperado pelo usuário.

Com base nas definições propostas por Dias (2008), é possível perceber que os defeitos estão presentes nos sistemas e podem ser causados pela má utilização de uma determinada tecnologia. Defeitos podem gerar erros. Ou seja, isso significa que o software não foi construído seguindo a sua especificação. Os erros podem causar comportamentos inesperados (falhas). Essas falhas afetam diretamente o a utilização do sistema, podendo até mesmo inviabilizar a utilização do mesmo.

2.2.4.1. Tipos de testes

Produtos que passam por um processo de engenharia podem ser testados de duas formas diferentes: conhecendo as funcionalidades que o produto deve oferecer, ou sabendo como essas funcionalidades foram implementadas. Dessa forma, existem dois grupos principais de testes:

- **Testes caixa preta** (*Black Box*): também conhecido como teste funcional, verificam as funcionalidades do software sem levar em consideração a lógica interna do componente.
- **Testes caixa branca** (*White Box*): também conhecido como teste estrutural, avaliam a lógica interna dos componentes, a forma como esses componentes foram implementados, ou seja, os elementos técnicos.

Os testes de software são executados em diferentes níveis durante todo o ciclo de vida do software. Alguns níveis de testes são: teste unitário (ou teste de unidade), teste de integração, teste de sistema, teste de aceitação e teste de regressão.

O objetivo de dividir os testes de software em níveis é garantir que o software seja testado em todas as etapas de desenvolvimento: desde funções isoladas (testes unitários) até a integração de módulos ou componentes.

O teste unitário é aplicado aos menores componentes de código criados (funções ou métodos, por exemplo). O objetivo desse teste é avaliar a implementação de um componente, visando garantir que este atende as suas especificações. De acordo com Pressman (2006, p.295),

> O teste de unidade focaliza o esforço de verificação na menor unidade de projeto do software [...]. Usando a descrição de projeto no nível de componentes como guia, caminhos de controle importantes são testados para descobrir erros dentro dos limites do módulo. A complexidade relativa dos testes e dos erros descobertos é limitada pelo escopo restrito estabelecido para o teste de unidade. O teste de unidade enfoca a lógica interna de processamento e as estruturas de dados dentro dos limites de um componente. Esse tipo de teste pode ser conduzido em paralelo para diversos componentes.

De acordo com Pressman (2006), os testes de unidade englobam os seguintes testes:

- **Interface**: procura identificar problemas para o recebimento ou envio de dados através da interface de um determinado módulo;
- **Estruturas lógicas de dados**: verifica a integridade dos dados durante a execução de um algoritmo;

- **Condições-limites**: procura garantir que o módulo não apresentará problemas ao operar nos limites estabelecidos para o processamento;
- **Caminhos independentes**: visa garantir que todos os caminhos da estrutura de controle foram executados pelo menos uma vez;
- **Caminhos de manipulação de erros**: todos os caminhos de manipulação de erros são testados.

O teste de integração visa garantir que os componentes de um sistema podem trabalhar em conjunto adequadamente. Pressman (2006, p. 297) afirma que o "teste de integração é uma técnica sistemática para construir a arquitetura do software enquanto, ao mesmo tempo, conduz testes para descobrir erros associados às interfaces".

Testes de sistema são executados em um sistema, ou subsistema, em um ambiente operacional controlado. O objetivo desse teste é verificar se todas as funcionalidades do sistema são executadas corretamente.

Segundo Pressman (2006), fazem parte dos testes de sistema os seguintes tipos de testes:

- **Teste de recuperação**: força o sistema a falhar para testar se o sistema é capaz de se recuperar de falhas causadas por diversos fatores e, se essa recuperação ocorre de forma eficiente;
- **Teste de segurança**: verifica se os mecanismos de proteção contra invasões do sistema estão funcionando adequadamente;
- **Teste de estresse**: "executa um sistema de tal forma que demanda recursos e quantidade ou volumes anormais" (PRESSMAN, 2006, p. 306). O objetivo é verificar qual o comportamento apresentado pelo sistema em situações como essa;
- **Teste de desempenho**: procura verificar qual o desempenho apresentado pelo sistema ao executá-lo em situações normais e em situações extremas, como as propostas pelo teste de estresse;

O objetivo do teste de aceitação é verificar se o sistema desenvolvido atende efetivamente às necessidades dos clientes. De acordo com Dias (2008, p.56), os testes de aceitação "são realizados geralmente por um restrito grupo de usuários finais do sistema. Esses simulam operações de rotina do sistema de modo a verificar se seu comportamento está de acordo com o solicitado."

O teste de regressão é realizado cada vez que algum tipo de modificação é feita no sistema. Um conjunto de testes previamente executado, deve ser refeito para garantir que o sistema continua operando como esperado. Dias (2008, p. 56) afirma que,

> Teste de regressão não corresponde a um nível de teste, mas é uma estratégia importante para redução de ‘efeitos colaterais’. Consiste em se aplicar, a cada nova versão do software ou a cada ciclo, todos os testes que já foram aplicados nas versões ou ciclos de teste anteriores do sistema. Pode ser aplicado em qualquer nível de teste.

Não é possível testar um software completamente e garantir que ele está livre de falhas. No entanto, o objetivo dos testes de software é garantir que a maior parte das falhas sejam encontradas e corrigidas. Dessa forma, melhorando a qualidade do software.

2.2.4.2. Processo de teste

O processo de teste deve se basear em metodologias que se adaptem ao processo de desenvolvimento de software. Esse processo deve contar com pessoas especializadas e ferramentas adequadas, para que possa ser realizado com sucesso.

Um aspecto importante que deve ser levado em consideração é a integração entre o processo de teste e o processo de desenvolvimento de software. As atividades de desenvolvimento e de teste devem ser conduzidas paralelamente. Segundo Dias (2008, p. 56) “o planejamento dos testes deve ocorrer em diferentes níveis e em paralelo ao desenvolvimento do software”, conforme ilustrado na Figura 9.

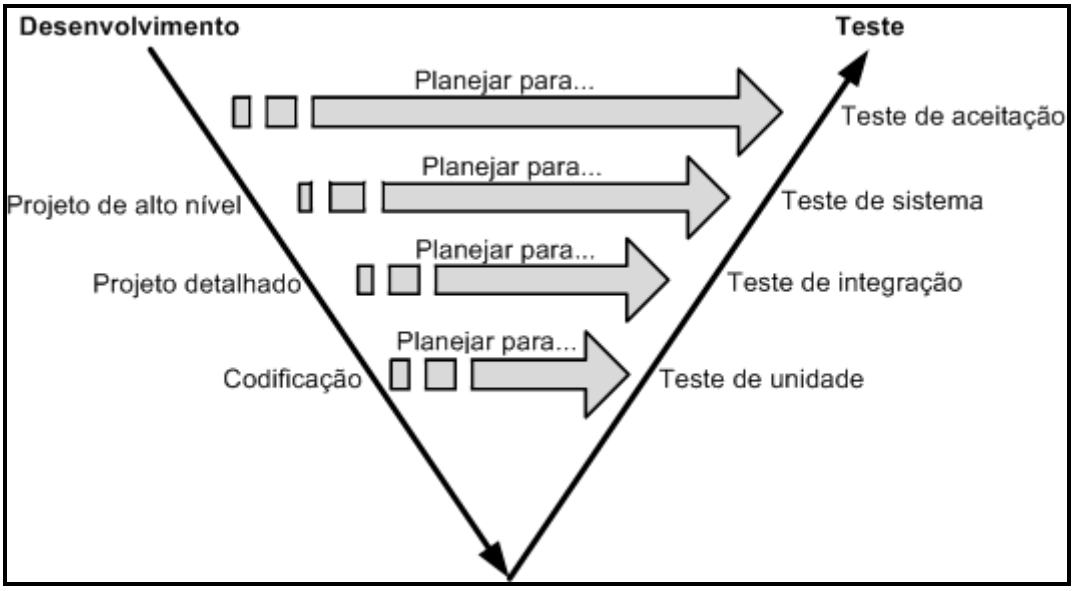

Figura 9 – Paralelismo entre as atividades de desenvolvimento e teste de software
Fonte: Graig e Jaskiel (2002 *apud* DIAS, 2008)

Moreira e Rios (2003) propõem as seguintes fases para um processo de teste:

- **Procedimentos iniciais**: elaboração de um guia operacional de testes (GOT). Esse documento contemplará os seguintes temas: objetivo do projeto de testes, pessoas envolvidas, responsabilidades de cada um, plano preliminar de trabalho, avaliação de riscos, e qualquer outro aspecto relevante às atividades de teste.
- **Planejamento**: elaboração de estratégias e planos de testes.
- **Preparação**: disponibilização do hardware, software e pessoas necessárias para executar os testes.
- **Especificação**: elaboração de casos e roteiros de testes, que representam quais e como serão os testes realizados, bem como a verificação da documentação do sistema.
- **Execução**: execução dos testes seguindo os casos e roteiros de testes previamente especificados.
- **Entrega**: conclusão e entrega dos testes realizados.

Outra proposta para o processo de teste é de Sommerville (2003). Essa proposta é composta das seguintes atividades:

1. Determinar o que será avaliado pelo teste;
2. Decidir como os testes devem ser realizados;
3. Desenvolver os casos de teste;
4. Determinar quais são os resultados esperados de cada teste;
5. Executar os testes;
6. Comparar os resultados obtidos com os resultados esperados.

2.2.5. Reengenharia de software

As organizações investem muito no desenvolvimento de software. Para que essas organizações obtenham o retorno desse investimento, é necessário que esses sistemas sejam utilizados durante muito tempo. Porém, conforme os anos vão passando, é necessário modificar o sistema, seja para corrigir erros, ou adicionar funcionalidades, ou outro motivo qualquer. Essas modificações acabam gerando novos requisitos para o software.

Sistemas legados incorporam as modificações feitas durante muitos anos em um software. Um sistema legado "é aquele que tenha sido desenvolvido de forma a satisfazer às necessidades antigas do cliente e que tenha sido implementado utilizando uma tecnologia antiga" (PETERS e PEDRYCZ, 2001, p. 559).

Ainda existem organizações que dependem desses tipos de sistemas. Essas organizações não podem simplesmente descartar esses sistemas e iniciar o desenvolvimento de sistemas mais modernos, devido aos riscos que uma atitude como essa representa. Pedrycz e Peters (2001) citam algumas razões que comprovam os riscos envolvidos na substituição de sistemas legados:

- Dificilmente existe uma especificação completa do sistema legado, portanto, não é possível desenvolver um novo sistema que contemple todas as funcionalidades do sistema legado.
- Os processos organizacionais e a forma como os sistemas legados operam, normalmente, estão interligados. Portanto, se o sistema for substituído, é provável que os processos organizacionais tenham que ser modificados, gerando conseqüências imprevisíveis.
- Muitas regras de negócio importantes podem estar inseridas no software legado, no entanto, essas regras podem não estar documentadas. Dessa forma, comprometer essas regras pode trazer conseqüências negativas à organização.
- O desenvolvimento de um novo software é um processo arriscado. Devido aos problemas que podem ocorrer durante o desenvolvimento, eles podem custar muito mais que o previsto e ser entregue muito depois do prazo estipulado.

Continuar a utilizar sistemas legados pode evitar os riscos de substituição, porém, a manutenção desses sistemas se tornará cada vez mais difícil. Isso ocorre, principalmente, pelo fato de que partes diferentes do sistema foram desenvolvidas por equipes diferentes, que utilizam estilos de programação diferentes. Outro motivo que dificulta a manutenção é o fato desses sistemas, normalmente, serem implementados utilizando linguagens de programação obsoletas. Além disso, eles não dispõem de documentação adequada, muitos anos de manutenção podem ter modificado a estrutura original do sistema, entre muitos outros fatores.

Uma alternativa à substituição de sistemas legados é a reengenharia de software. A reengenharia de software se propõe:

> Reimplementar sistemas legados para que sua manutenção seja mais fácil. A reengenharia pode envolver redocumentar, organizar e reestruturar o sistema, traduzir o sistema para uma linguagem de programação mais moderna e modificar e atualizar a estrutura e os valores dos dados do sistema. (SOMMERVILLE, 2003, p. 533)

Do ponto de vista técnico, a reengenharia pode ser considerada como uma técnica de segunda classe, já que ela não altera as funcionalidades e, muitas vezes, nem a arquitetura do

software. Porém, do ponto de vista dos negócios, essa pode ser a única solução viável para continuar utilizando sistemas legados. Porém, Pressman (2006, p. 685), afirma que:

> A reengenharia leva tempo; tem um custo significativo em dinheiro, e absorve recursos que poderiam, por outro lado, ser usados em necessidades imediatas. Por todas essas razões, a reengenharia não é conseguida em poucos meses ou mesmo em poucos anos. A reengenharia de sistemas de informação é uma atividade que absorve recursos de tecnologia da informação durante muitos anos.

Os principais benefícios alcançados pela reengenharia aplicada a sistemas legados são:

- Aumento da manutenibilidade do software;
- Identificação de candidatos à reutilização;
- Valorização do sistema que está passando pelo processo de reengenharia;
- Aprimoramento da gestão de riscos do projeto.

A Figura 10 demonstra as atividades do processo de reengenharia.

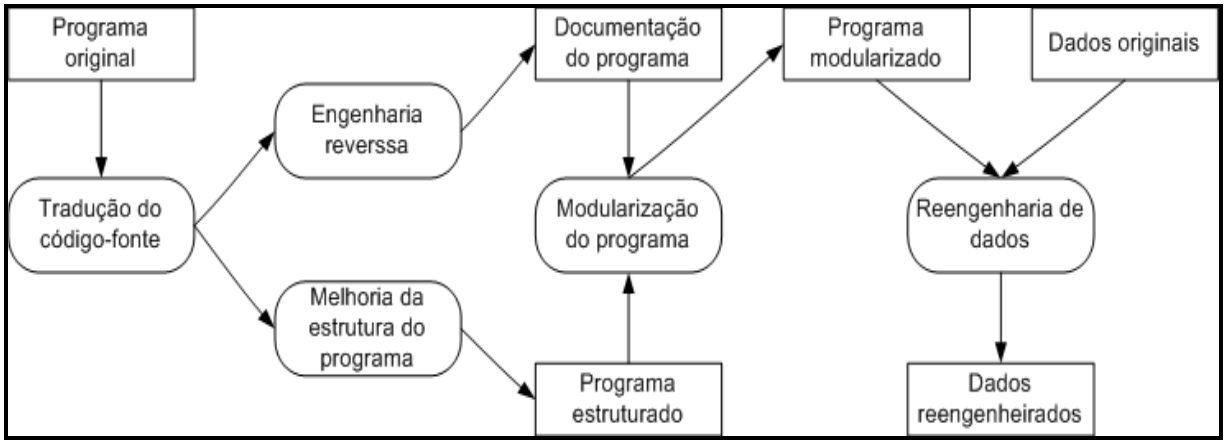

Figura 10 – Processo de reengenharia
Fonte: Sommerville (2003)

De acordo com Somerville (2003), as atividades da reengenharia de software são:

1. **Tradução do código fonte**: é feita a conversão para uma versão mais moderna da linguagem ou para uma nova linguagem;
2. **Engenharia reversa**: analisa-se o sistema legado para identificar e recuperar as informações referentes ao sistema;
3. **Melhoria da estrutura do programa**: analisa-se e modifica-se a estrutura do programa para torná-lo mais fácil de ser compreendido;
4. **Modularização do programa**: agrupam-se as partes relacionadas do sistema;
5. **Reengenharia de dados**: modificam-se os dados processados pelo sistema, para que estes reflitam as mudanças feitas no sistema.

O processo de reengenharia não precisa envolver todas essas atividades. Por exemplo, nem sempre é necessário migrar o código fonte para outra linguagem de programação, caso exista documentação adequada, a engenharia reversa se torna desnecessária.

É necessário analisar quais são as reais necessidades de uma organização antes de decidir entre efetuar a substituição ou a reengenharia de sistemas legados. Pois, ambas as abordagens podem trazer benefícios, mas também podem causar muitos danos à organização.

2.3. Arquitetura de software

Com o passar do tempo, os softwares foram se tornando cada vez maiores e mais complexos. Para tentar minimizar os problemas encontrados durante o desenvolvimento de software, os engenheiros de software utilizam princípios de projeto, como a ocultação de informação. No entanto, foi preciso encontrar meios para obter resultados de baixo custo e alta qualidade. Foi neste contexto que a arquitetura de software surgiu, para tentar minimizar os problemas encontrados durante o desenvolvimento de software definindo estruturas que possam ser re-utilizadas em outros projetos

A arquitetura de software estuda a organização global dos sistemas de software e os relacionamentos existentes entre os subsistemas e componentes. Bass, Clements e Kazman (2003 *apud* PRESSMAN, 2006), definem o termo arquitetura de software da seguinte forma: “a arquitetura de software de um programa ou sistema computacional é a estrutura ou estruturas do sistema que abrange os componentes de software, as propriedades externamente visíveis desses componentes e as relações entre eles”.

Segundo Ahmed e Umrysh (2002), qualquer definição de arquitetura de software faz referência a alguns dos seguintes itens:

- Estruturas estáticas do software, ou seja, a forma como os elementos do software se relacionam;
- Estruturas dinâmicas do software, ou seja, as relações que determinam como o software fica quando executado;
- Composição (ou decomposição) do software, se refere aos componentes do software, como subsistemas ou módulos;
- Camadas e interações existentes entre elas;
- Organização das partes físicas do software a serem distribuídas, como arquivos *.jar*, *.war* ou *.exe*;

- Estilos que guiam o desenvolvimento e a evolução do software;
- Funcionalidades do software;
- Decisões relacionadas à organização do sistema de software;
- Considerações relacionadas à reutilização, desempenho, etc.

A arquitetura não é um software operacional, ao contrário, a arquitetura é uma representação que permite:

- Analisar se o projeto atende aos seus requisitos;
- Considerar alternativas de projeto em um estágio em que ainda é possível fazer mudanças sem grandes conseqüências;
- Reduzir os riscos encontrados na construção de software.

Arquiteturas bem sucedidas apresentam algumas características fundamentais. Essas características afetam diretamente todos os requisitos não-funcionais de um software. Algumas características são:

- **Eficiência**: refere-se aos recursos necessários para que o software realize suas tarefas de forma adequada;
- **Integridade**: pode ser tratada em termos de unificação do projeto em níveis distintos. "A arquitetura de software descreve a organização de um sistema de software num nível mais elevado, objetivando a unificação do projeto, ou seja, ela serve de referência para o projeto". (MENDES, 2002, p.66);
- **Flexibilidade**: refere-se ao esforço necessário para modificar um sistema de software, ou seja, a facilidade que o sistema oferece ao ser modificado.

2.3.1. Estilos e padrões arquiteturais

Os estilos arquiteturais possuem características bem definidas. Conhecer estilos arquiteturais diminui o esforço para entender o projeto de sistemas desenvolvido por outras pessoas.

Os estilos arquiteturais fornecem suporte a um conjunto de requisitos não-funcionais. Definir qual estilo é mais apropriado a um sistema depende da definição de quais são os principais requisitos funcionais e não-funcionais que o sistema deve atender. Para Pressman (2006, p. 211),

> Cada estilo descreve uma categoria de sistema que abrange (1) um conjunto de componentes (um banco de dados, módulos computacionais) que realizam a função que é requisito do sistema; (2) um conjunto de conectores que possibilita

> "comunicação, coordenação e cooperação" entre os componentes; (3) restrições que definem como os componentes podem ser integrados para formar o sistema; e (4) modelos semânticos que possibilitam ao projetista entender as propriedades gerais de um sistema pela análise das propriedades conhecidas de suas partes constitutivas.

Padrões arquiteturais definem uma abordagem específica para tratar de algumas características comportamentais do sistema. Padrões arquiteturais podem ser utilizados em conjunto com um estilo arquitetural, definindo a estrutura global de um sistema. Pressman (2006, p.211) afirma que:

> O padrão difere do estilo em um certo número de modos fundamentais: (1) o escopo de um padrão é menos amplo, enfoca um aspecto da arquitetura em vez de toda a arquitetura; (2) um padrão impõe uma regra de arquitetura, descrevendo como o software vai manipular em algum tópico de sua funcionalidade em termos de infra-estrutura (por exemplo concorrência); (3) padrões arquiteturais tendem a atender tópicos comportamentais específicos no contexto arquitetural, por exemplo, como uma aplicação de tempo real manipula sincronização ou interrupções.

A Tabela 4 conceitua alguns padrões arquiteturais mais utilizados.

Tabela 4 – Padrões arquiteturais

Padrão arquitetural	**Descrição**
Concorrência	São utilizados quando uma aplicação precisa tratar múltiplas tarefas simultaneamente. Existem diversas maneiras pelas quais uma aplicação pode tratar a concorrência e cada uma pode ser tratada por um padrão arquitetural diferente.
Persistência	Dados persistentes são armazenados em bancos de dados ou arquivos para que possam, posteriormente, ser manipulados. Normalmente são utilizados dois padrões arquiteturais para persistência: sistemas de gestão de banco de dados e persistência de aplicação.
Distribuição	Tratam a maneira pela qual os sistemas ou componentes de sistemas se comunicam em um ambiente distribuído. Esse tipo de padrão se preocupa com a maneira como os componentes se comunicam entre si e o tipo de comunicação existente entre eles. Mendes (2002, p.27), define a arquitetura distribuída "como um programa concorrente no qual os processos se comunicam por meio da passagem de mensagens".

Fonte: Kamyla Estuqui Parrado (2008).

Nem todos os autores concordam com a distinção feita entre estilos e padrões arquiteturais. Alguns autores usam esses termos como sinônimos, enquanto que outros fazem esta distinção.

De acordo com Mendes (2002), alguns tipos de arquiteturas mais comuns são os seguintes:

- **Arquitetura centrada nos dados**: também conhecido como quadro-negro ou *blackboard*, este estilo arquitetural possui um depósito de dados que fornece acesso freqüente a outros componentes que atualizam, adicionam ou retiram os dados contidos no depósito (MENDES, 2002). Nesta arquitetura também existe um conjunto de células de conhecimento que são logicamente independentes. O banco de dados compartilhado dessa arquitetura possui uma organização independente da aplicação. Devido a isso, a manutenibilidade é uma das principais vantagens dessa arquitetura.
- **Arquitetura de fluxo de dados**: também conhecido como *pipes* e filtros, é utilizado quando os dados de entrada devem ser transformados em dados de saída. Esse estilo considera a "existência de uma rede pela qual flui dados de uma extremidade (origem) à outra (destino). O fluxo de dados se dá através de *pipes* e os dados sofrem transformações quando processados nos filtros" (MENDES, 2002, p.12). Esse estilo é adequado para projetos que necessitem de vários estágios de processamento.
- **Arquitetura de chamada e retorno**: Esse estilo arquitetural fornece uma estrutura fácil de modificar e ampliar e pode ser subdividido em duas categorias:
    - **Arquiteturas de programa principal/subprograma**: "decompõe a função em uma hierarquia de controle em que um programa 'principal' aciona um certo número de componentes de programa, que, por sua vez, invocam outros componentes" (PRESSMAN, 2006, p.213).
    - **Arquiteturas de chamada de procedimentos remotos**: "mecanismo de transferência de controle de um processo (chamador) para outro (chamado), sendo depois o controle retornado ao processo que emitiu a chamada" (MENDES, 2002, p.29).
- **Arquitetura orientada a objetos**: nesse estilo arquitetural, os componentes de um sistema encapsulam os dados e as operações que atuam sobre esses dados. A comunicação ocorre através da troca de mensagens. O estilo arquitetural de objetos tem como base a utilização de tipos abstratos de dados. Mendes (2002) destaca algumas propriedades importantes deste estilo:

- Objetos são entidades independentes que podem sofrer modificação, já que toda informação é mantida no próprio objeto;
- É possível compreender melhor o sistema através do mapeamento das entidades reais e dos objetos que atuam no controle do sistema;
- Os objetos se comunicam através de troca de mensagens;
- Objetos são independentes e, por isso, podem ser reutilizados;
- Dependendo de como foram projetados, os objetos podem estar distribuídos e executar seqüencialmente ou em paralelo.

- **Arquitetura em camadas**: este estilo arquitetural define um certo número de camadas diferentes, cada uma realizando operações que se tornam progressivamente mais próximas do conjunto de instruções de máquina. As camadas exteriores tratam as operações de interface com o usuário. As camadas mais internas realizam a interface com o sistema operacional. As camadas intermediárias fornecem serviços e funções de diversos tipos (MENDES, 2002). Cada camada adiciona um nível de abstração sobre a camada anterior. Os diversos níveis de abstração fornecidos neste estilo arquitetural aumentam a flexibilidade do sistema. No entanto, o desempenho pode ser comprometido, já que uma solicitação precisará passar por diversas camadas até ser tratada.
- **Arquiteturas distribuídas**: são compostas por múltiplos processadores que utilizam mecanismos de mensagem para se comunicar. São compostos por filtros (transformam os dados), clientes (processo que inicia algum tipo de atividade), servidor (processa as solicitações de clientes) e pares (ou *peers*, conjunto de processos que oferecem algum tipo de serviço). Mendes (2002), afirma que as arquiteturas distribuídas podem ser subdivididas em:
  - **Arquiteturas de multiprocessadores**: composta por vários processos sendo executados em processadores diferentes.
  - **Arquiteturas cliente-servidor**: conjuntos de serviços são oferecidos por servidores. Esses serviços são consumidos por clientes que precisam saber quais são os servidores disponíveis, mas não precisam saber da existência de outros clientes.
  - **Arquiteturas de objetos distribuídos**: nesse tipo de arquitetura distribuída, um conjunto de serviços é oferecido por objetos que fornecem uma interface para acessar tais serviços. Não é feita nenhuma distinção entre quem solicita e quem fornece os serviços.

2.3.2. Padrões para arquiteturas distribuídas

O desenvolvimento de arquiteturas distribuídas exige que exista uma padronização entre as interfaces de comunicação utilizadas. Isso é necessário para garantir a interoperabilidade entre os sistemas.

Existem diversos padrões para implementação de arquiteturas distribuídas, entre esses padrões se destacam SOA (*Service Oriented Architecture*), CORBA (*Common Object Request Broker Architecture*) e RMI (*Remote Method Invocation*).

Kodali (2005, p. 1), define SOA como:

> Uma evolução da computação distribuída baseada no paradigma requisição/resposta para aplicações síncronas e assíncronas. A lógica de negócio da aplicação em funções individuais são modularizadas e distribuídas como serviços para aplicações clientes/consumidoras. O ponto chave dos serviços é a sua natureza fracamente acoplada, ou seja, a interface do serviço é independente da sua implementação.

Ao utilizar SOA as funcionalidades de uma aplicação são fornecidas como serviços. Um serviço, no contexto SOA, representa uma função do sistema que pode ser acessada (consumida) por outros sistemas através de interfaces de comunicação. Uma das vantagens da utilização de SOA, é que a interface através da qual um serviço é disponibilizado é totalmente independente de sua implementação. Dessa forma, não é necessário conhecer os detalhes de implementação para utilizar os serviços fornecidos por um sistema. Devido a isso, um serviço pode ser implementado em *Java* e consumido por aplicações construídas com a plataforma *Microsoft .Net*.

O objetivo principal do SOA é a interoperabilidade entre sistemas. Ou seja, garantir que sistemas diferentes consigam trabalhar em conjunto através da utilização de interfaces e protocolos bem definidos.

Atualmente, a forma mais utilizada para implementar SOA é através de *web services*. Booth *et al.* (2004, p. 7), definem um *web service* como "sistema de software projetado para suportar a interoperabilidade entre máquinas em uma rede". Os *web services* enviam e recebem dados através de arquivos XML[2]. Ou seja, não importa em qual plataforma um *web service* (ou a aplicação consumidora) foi construída, pois, as informações, ao serem enviadas ou recebidas, são convertidas para XML.

No mercado existem dois padrões principais para *web services*: *Web Services WS-** e *Web Services RESTful*.

[2] *eXtensible Markup Language*.

Os *web services WS-** utilizam uma série de tecnologias e protocolos. Kodali (2005) cita alguns desses protocolos:

- **WSDL (*Web Services Description Language*)**: linguagem baseada em XML utilizada para descrever *web services*.
- **UDDI (*Universal Description, Discovery and Integration*)**: padrão utilizado para publicar/distribuir *web services*.
- **SOAP (*Simple Object Access Protocol*)**: protocolo utilizado para o envio/recebimento de mensagens com *web services* WS-*. Ou seja, o SOAP representa a camada de transporte entre a aplicação consumidora e os serviços.

Bairos e Pereira (2008, p. 26), afirmam que "o consórcio de empresas que participou da definição dos padrões WS-* elaborou especificações para cenários bastantes diversificados." O consórcio desenvolveu padrões para todas as situações que pudessem surgir. Com isso, apenas quem participa ativamente do consórcio tem domínio sobre todas as especificações. Os desenvolvedores de aplicação utilizam apenas um subconjunto das especificações existentes.

Os *web services RESTful* originaram-se da tese de doutorado de *Roy Fielding*, um dos principais criadores do HTTP[3] e fundador do *Apache Software Foundation*. Segundo Fielding (2000, p. 86),

> O estilo *Representational State Transfer* (REST) é uma abstração dos elementos da arquitetura de sistemas de hipermídia distribuída. REST ignora os detalhes da implementação dos componentes e a sintaxe dos protocolos para se focar nas regras dos componentes, nas restrições sob suas interações com outros componentes e sua interpretação dos elementos de dados importantes. Ele inclui as restrições fundamentais sobre componentes, conectores e dados que definem a base para a arquitetura *web*, e a essência para este comportamento são as aplicações baseadas em rede.

Os *web services RESTful* utilizam basicamente duas tecnologias: XML e HTTP. O XML (*eXtensible Markup Language*) é a linguagem universal de comunicação para qualquer tipo de *web service*, ou seja, todos os dados enviados/recebidos estão sempre em arquivos XML. Uma mensagem HTTP contém toda a informação necessária para solicitar um serviço. Uma grande vantagem dos *web services RESTful* é a simplicidade de suas regras, se comparadas com as especificações dos *web services WS-**.

Um conceito importante em REST é a existência de recursos. Cada recurso possui um identificador, que, no HTTP, é chamado de *Uniform Resource Identifier* (URI). Um URI representa o "elemento mais simples da arquitetura web e o mais importante" (FIELDING,

[3] *Hypertext Transfer Protocol*. Protocolo utilizado para transmissão de dados na Internet.

2000, p.109). O URI indica o que será manipulado enquanto o método HTTP[4] determina como será manipulado. O URI deve conter toda a informação necessária para identificar o recurso que será manipulado.

É importante ressaltar que *web services* não são SOA. Ou seja, "*web services* tratam de especificações de tecnologia, enquanto que SOA é um padrão de software." (NATIS, 2003 *apud* KODALI, 2005, p. 4). Kodali (2005, p.4) afirma que "SOA é um padrão arquitetural, enquanto *web sevices* são serviços implementados utilizando um conjunto de padrões". Dessa forma, os *web sevices* representam apenas uma das maneiras através da qual SOA pode ser implementada.

CORBA (*Common Object Request Broker Architecture*) foi desenvolvido pelo OMG[5] para efetuar troca de objetos entre sistemas distribuídos. Segundo a especificação CORBA (OMG, 1999), os principais componentes desta arquitetura são:

- ***Object Request Broker* (ORB)**: representa a base para construir aplicações distribuídas e para a interoperabilidade entre elas. Permite que objetos recebam e enviem requisições e resposta em ambientes distribuídos.
- ***Object Services***: os serviços são necessários para construção de qualquer sistema distribuído. Em CORBA, os *object services* são "coleções de serviços (interfaces e objetos) que suporta as funcionalidades básicas para usar e implementar objetos" (OMG, 1999, p. xxviii).
- ***Common Facilities***: de acordo com a OMG (1999), são "coleções de serviços que muitas aplicações podem acessar, mas não são *object services* fundamentais".
- ***Application objects***: são produtos controlados por um fornecedor específico. Não são padronizados pela OMG.

O ORB é a parte mais importante do CORBA. O ORB está entre o cliente e o objeto. É o ORB que aceita uma requisição, envia a requisição para o objeto e, quando a resposta está disponível, envia para o cliente. O ORB é o responsável por:

> Todos os mecanismos necessários para encontrar a implementação do objeto para a requisição, para preparar a implementação do objeto para receber a requisição e preparar os dados para se comunicar com a requisição. A interface vista pelo cliente é completamente independente de onde os objetos estão localizados, de qual linguagem de programação eles foram implementados, ou qualquer outro aspecto que não seja refletido na interface do objeto (OMG, 1999, p.22).

[4] Exemplos de métodos HTTP são: *GET*, *POST*, *DELETE* e *PUT*.

[5] *Object Management Group*. Organização internacional responsável pela padronização de diversas tecnologias, incluindo CORBA e UML.

CORBA utiliza uma linguagem declarativa baseada em *C* conhecida como IDL (*Interface Definition Language*) para definir as interfaces dos objetos. A OMG definiu padrões desta linguagem para *C*, *C++*, *Java*, *COBOL*, *Smalltalk*, *Ada*, *Lisp*, *Python*, entre outras.

O *Remote Method Invocation* (RMI) é um método desenvolvido para permitir que objetos ativos em uma máquina virtual *Java* interajam com objetos ativos em outras máquinas virtuais *Java*. Essa tecnologia é limitada a *Java*, ou seja, apenas aplicações que sejam executadas em uma máquina virtual *Java* podem utilizar RMI.

2.3.3. Arquitetura MVC (*Model*, *View*, *Controller*)

A arquitetura MVC (*Model*, *View*, *Controller*) se baseia em agentes. Agentes possuem estados e são capazes de reagir a eventos. Nesta arquitetura um agente é visto sob três perspectivas: *model*, *view* e *controller*.

Segundo Mendes (2002, p. 132),

> O componente *model* define a competência do agente, ou seja, seu núcleo funcional. O componente *view* define o comportamento perceptivo do agente para saída, fornecendo suporte à apresentação. O componente *controller* denota o comportamento perceptivo do agente para entrada. Ele é encarregado de tratar as entradas e comandos de usuários bem como gerenciar toda a informação trocada entre o usuário e o componente *model*. Na arquitetura MVC, o componente *model* comunica-se diretamente com os componentes *view* e *controller*.

O *model* representa os objetos estruturais de um sistema. Ele simula o domínio da aplicação. O componente *view* representa a interface gráfica com a qual o usuário interage. É através da *view* que o usuário insere dados no sistema e visualiza o resultado do processamento desses dados. O componente *controller* é o responsável pelo tratamento dos eventos solicitados pelo usuário. Uma das vantagens oferecidas por esta arquitetura é a modularidade. Isso permite que partes de uma aplicação sejam reutilizadas em outras aplicações.

2.3.4. Requisitos arquiteturais

Durante as fases iniciais do processo de desenvolvimento de um sistema, é definido um conjunto de requisitos. Esses requisitos podem ser vistos como a especificação do que o

sistema deve fazer (requisitos funcionais) e como o sistema deve fazer (requisitos não-funcionais).

Quando a arquitetura é o fator que orienta o processo de desenvolvimento de software, os requisitos associados à arquitetura devem ser colocados em primeiro plano. A complexidade de um sistema de software é definida tanto por seus requisitos funcionais quanto por requisitos não-funcionais. Conforme o tamanho e a complexidade dos sistemas aumentam, o suporte a requisitos não-funcionais depende das decisões tomadas no projeto arquitetural do sistema.

Os atributos de projeto são definidos por Mendes (2002, p.39) "como princípios que norteiam o processo de desenvolvimento de software, visando a obter um produto final que satisfaça aos requisitos identificados no início do processo".

Os requisitos não-funcionais têm um papel muito importante durante o desenvolvimento de um sistema. Esses requisitos podem ser utilizados como critérios para escolhas de alternativas de projeto, estilo arquitetural e forma de implementação.

2.3.4.1. Modularidade

Sistemas muito grandes normalmente são divididos em partes menores, denominadas módulos. Uma das grandes vantagens deste atributo de projeto é permitir lidar com os detalhes de cada módulo isoladamente. A modularidade também permite lidar com características comuns a todos os módulos e as relações existentes entre eles de modo a integrá-lo em um sistema.

A separação de interesses permite lidar com diversos aspectos de um problema concentrando-se em cada aspecto isoladamente. Segundo Mendes (2002), a separação de interesses é aplicada quando um sistema é decomposto em vários módulos, permitindo isolar uma parte da funcionalidade do sistema de outras.

Para Mendes (2002), a capacidade de decompor um sistema é baseada na idéia de dividir um problema maior em um conjunto de subproblemas menores e executar este procedimento, recursivamente, a cada subproblema.

De acordo com Mendes (2002), a modularidade tem três objetivos:

- Decompor um sistema complexo e/ou de grande porte;
- Compor um sistema a partir de um conjunto de módulos;
- Compreender a modularização do sistema.

2.3.4.2. Usabilidade

Softwares são produtos utilizados por usuários. Devido a isso, eles precisam ser simples e fáceis de utilizar. Os softwares devem permitir ao usuário executar suas tarefas de forma intuitiva. O software, como qualquer outro produto, deve ser eficiente para atender as expectativas dos usuários.

A usabilidade está relacionada à facilidade que os usuários têm para aprender e utilizar o sistema. Os requisitos de usabilidade "especificam tanto o nível de desempenho quanto a satisfação do usuário no uso do sistema" (MENDES, 2002, p.46). Ou seja, a usabilidade está associada a:

- **Facilidade de aprendizado**: diz respeito ao tempo e ao esforço mínimo exigido para alcançar um determinado nível de desempenho na utilização do sistema.
- **Facilidade de uso**: diz respeito à velocidade de execução de tarefas e à redução de erros no uso do sistema.

De acordo com Mendes (2008, p. 25),

> Usabilidade é uma palavra que tem feito cada vez mais parte do vocabulário dos projetistas de sistemas de software. A usabilidade é um conceito chave no campo da Interação Humano-Computador (IHC), sendo um atributo de qualidade de sistemas que são fáceis de usar e de aprender. Em outras palavras, diz quão intuitiva é a interface gráfica de usuário ou, simplesmente, interface de usuário. Trata-se, portanto, de uma característica pela qual o usuário expressa seu interesse ou não em utilizar um sistema. Na grande maioria dos casos, os usuários preferem um sistema de fácil uso, mesmo com funcionalidades mais simples, a um sistema recheado de funcionalidades, porém de manipulação complexa e não intuitiva.

A usabilidade é um ponto decisivo para o sucesso de um produto. Quanto maior for à usabilidade de um sistema, menos o cliente terá que investir em treinamentos e maior será a eficiência com a qual as atividades a serão realizadas.

2.3.4.3. Manutenibilidade

A manutenção de software refere-se às modificações feitas no sistema após este ter sido disponibilizado para uso. A manutenção envolve tanto as atividades realizadas para corrigir falhas, quanto às relacionadas à evolução do sistema. Portanto, a manutenibilidade pode ser definida como a facilidade de adicionar ou modificar funcionalidades.

De acordo com Mendes (2002, p.48),

> Uma arquitetura de software define os componentes e conexões entre estes e, portanto, também define sob que circunstâncias componentes ou conectores podem ser alterados. Dessa forma, uma arquitetura ou estilo arquitetural de um sistema de software deveria efetivamente acomodar as modificações que precisarem ser feitas tanto durante seu desenvolvimento quanto após o sistema entrar em operação.

O software está em constante evolução, devido a isso, a manutenibilidade é um dos requisitos mais importante de qualquer sistema de software. Se o software não suportar as mudanças que surgirão com o passar do tempo, este tende a se tornar desatualizado. Conseqüentemente, chegará um ponto em que outros requisitos não-funcionais serão afetados e o software não atenderá adequadamente as necessidades de seus usuários.

Fielding (2000) classifica a manutenibilidade em diferentes aspectos:

- **Capacidade de evolução**: representa o grau em que os componentes de um sistema podem ser alterados sem impactar negativamente em outras partes do sistema;
- **Capacidade de extensão**: representa a facilidade para adicionar novos componentes ao sistema sem impactar no restante do sistema;
- **Capacidade de customização**: se refere à habilidade de especializar o comportamento de um elemento arquitetural para atender à necessidade de um cliente especifico.

2.3.4.4. Reusabilidade

A reusabilidade pode ser definida como a capacidade que um sistema, ou componente, possui de ser utilizado em diversos sistemas sem precisar ser modificado. Para que isso seja possível é preciso garantir um fraco acoplamento entre os componentes de um sistema.

De acordo com Fielding (2000, p.35), reusabilidade é a "propriedade da arquitetura de uma aplicação e seus componentes, conectores ou dados poderem ser reutilizados sem modificar outras aplicações".

A reusabilidade pode ser vista sob diferentes perspectivas. Ela pode ser orientada a componentes, processos ou conhecimento de um domínio. Mendes (2002) cita alguns exemplos de reutilização de componentes de software:

- **Aplicação**: toda a aplicação pode ser reutilizada.
- **Subsistemas**: os subsistemas de uma aplicação podem ser reutilizados.
- **Objetos ou módulos**: componentes de um sistema, englobando um conjunto de funções, podem ser reutilizados.
- **Funções**: componentes de software, que implementam uma única função podem ser reutilizados.

2.3.4.5. Confiabilidade e disponibilidade

A confiabilidade de software está relacionada à probabilidade do software não causar uma falha num sistema, durante um período de tempo sob condições especificadas. Em outras palavras, "a confiabilidade de software, geralmente definida em termos de comportamento estatístico, é a probabilidade de que o software operará como desejado num intervalo de tempo conhecido" (MENDES, 2002, p.48). Mais especificamente, sistemas confiáveis fornecem serviços que serão executados conforme especificado.

Algumas métricas são utilizadas para avaliar a confiabilidade de software. Segundo Mendes (2002), algumas dessas métricas são:

- **Taxa de ocorrência de falhas**: freqüência na qual o sistema falha em oferecer um serviço como esperado pelo usuário, ou seja, a freqüência na qual ocorre algum comportamento inesperado.
- **Probabilidade de falha durante fase operacional**: probabilidade de que o sistema se comporte de maneira inesperada ao entrar em operação.
- **Tempo médio até a ocorrência de falha ou *Mean Time to Failure* (MTTF)**: medida de tempo entre as falhas observadas. Oferece um indicativo de quanto tempo o sistema permanecerá operacional antes que uma falha aconteça.

Sommerville (2003, p. 305), define a disponibilidade como "a probabilidade de um sistema, em determinado instante, ser operacional e capaz de fornecer os serviços requeridos". Ou seja, a disponibilidade de um sistema garante que um os serviços solicitados pelos usuários estarão disponíveis no momento em que forem solicitados.

2.3.4.6. Escalabilidade

Os requisitos de um sistema mudam constantemente. Dessa forma, sistemas que se iniciam como pequenos projetos que visam atender a uma necessidade primária podem crescer além do esperado.

A escalabilidade de um sistema se refere a "habilidade de uma arquitetura suportar um grande número de componentes ou interações entre esses componentes com as configurações atuais" (FIELDING, 2000, p. 32). Ou seja, a escalabilidade de um sistema representa a capacidade que este tem de crescer e suportar novos componentes, sem que sejam necessárias mudanças drásticas em sua estrutura.

2.3.4.7. Portabilidade

A portabilidade diz respeito à facilidade de transferir um sistema de uma plataforma computacional para outra. Sistemas de software são portáveis quando podem ser executados em diversos tipos de ambiente sem que sejam necessárias grandes mudanças no sistema.

Conforme Mendes (2002, p.52), "a portabilidade de um componente ou sistema de software é proporcional à quantidade de esforço necessário para que funcione num novo ambiente". Ou seja, quanto menor for o esforço necessário para modificar um sistema, mais portável o sistema será.

A portabilidade pode ser apresentada de duas formas:

- **Binária**: ocorre quando os arquivos executáveis de um software podem ser transferidos de uma plataforma para outra sem a necessidade de compilar o código-fonte original novamente.
- **Código-fonte**: ocorre quando um software é desenvolvido em uma linguagem de programação que é suportada por diversas plataformas de hardware e software. Quando existe a necessidade transferir o software para uma nova plataforma, o código fonte é compilado novamente para gerar arquivos executáveis que possam ser executados nessa nova plataforma.

2.3.4.8. Desempenho

O desempenho de um sistema está ligado a sua usabilidade, pois, se um sistema for lento, certamente ele reduzirá a produtividade de seus usuários. O desempenho depende das iterações existentes entre os componentes de um sistema de software. Ou seja, os meios de comunicação utilizados pelos componentes do sistema afetam diretamente o desempenho do sistema.

Segundo Mendes (2002), os requisitos de desempenham restringem a velocidade dos sistemas de software. Isso pode ser visto em termos de:

- **Requisitos de resposta**: é o tempo de resposta de um sistema aceitável para os usuários;
- **Requisitos de processamento (*throughput*)**: especificam a quantidade de dados que deveriam ser processados num determinado período de tempo;

- **Requisitos de temporização**: específica a velocidade que o sistema deveria coletar os dados de entrada antes que estes sejam sobrescritos;
- **Requisitos de espaço**: refere-se ao uso da memória principal ou da memória secundária.

2.3.4.9. Segurança

A segurança representa a garantia de que qualquer tipo de acesso não-autorizado ao sistema será bloqueado. Ela representa o grau em que a arquitetura de uma aplicação está suscetível a falhas. Segundo Mendes (2002), existem diferentes ênfases relacionadas à segurança, algumas delas são:

- **Disponibilidade**: garantia de que o sistema estará disponível para uso sem qualquer tipo de interrupção dos serviços oferecidos;
- **Integridade**: assegura que acessos ou atualizações não-autorizadas não ocorram;
- **Confidencialidade**: garante que informações não serão disponibilizadas sem a devida autorização.

2.3.5. Análise arquitetural

A análise arquitetural abrange um conjunto de atividades que tem como objetivo produzir uma arquitetura adequada a um determinado sistema de software. Existem diversas abordagens para realização da análise arquitetural, no entanto, segundo Mendes (2002) algumas atividades são comuns a todas:

- **Desenvolvimento de modelos arquiteturais**: representa opções de arquiteturas construídas através do levantamento de requisitos do sistema;
- **Melhoria e síntese de uma solução**: incrementa as características do modelo arquitetural escolhido para o sistema;
- **Análise da solução**: identifica a necessidade de refinar o modelo arquitetural escolhido.

Mendes (2002, p.65) afirma que “a análise de uma arquitetura de software é um processo iterativo no qual são feitos refinamentos de um modelo arquitetural inicial derivado a partir de um conjunto de requisitos arquiteturais.”

2.3.5.1. Cenários de qualidade

Cenários são exemplos de interações entre usuários e o sistema. A construção de cenários de qualidade permite simular situações de interação entre os usuários e o sistema. Dessa forma é possível avaliar se o sistema atende aos requisitos arquiteturais especificados e, principalmente, atende as necessidades dos usuários. O tipo de cenário desenvolvido depende do tipo de sistema ao qual o cenário está associado.

Mendes (2002) descreve um cenário como sendo uma história que explica como o sistema deve funcionar, ou seja, como os usuários interagem com o sistema.

Após a identificação de algumas das características e funcionalidades do sistema, é possível começar a construir cenários para identificar outros requisitos do sistema e verificar a necessidade de modificações ou adaptações no software. Os cenários desenvolvidos permitem avaliar se a arquitetura de software inicialmente escolhida atende, ou não, aos requisitos do sistema.

A validação é uma das etapas do processo desenvolvimento de uma arquitetura de software e visa à validação dos cenários de qualidade desenvolvidos. A validação pode ser realizada através de discussões envolvendo os principais influenciadores do sistema. Mendes (2002, p. 62), afirma que um dos principais objetivos da validação "é fazer o mapeamento de cenários em requisitos arquiteturais do sistema". Dessa forma, os cenários de qualidade podem ser vistos como uma maneira de identificar os requisitos arquiteturais de um sistema.

2.3.5.2. SAAM (*Software Architecture Analysis Method*)

SAAM (*Software Architecture Analysis Method*) é um método de análise arquitetural inicialmente projetado para auxiliar a comparação entre alternativas arquiteturais. A proposta inicial consistia em:

- Definir cenários que representassem as interações mais importantes entre o sistema e os principais influenciadores do mesmo;
- "Utilizar cenários para gerar uma partição funcional do domínio, uma ordenação parcial das funções e um acoplamento dos cenários às várias funções existentes na partição" (MENDES, 2002, p.67);
- Utilizar a partição funcional, a ordenação parcial e os cenários para realizar a análise das alternativas propostas.

De acordo com Mendes (2002), esse método é composto por cinco passos interdependentes:

- **Desenvolvimento de cenários**: ilustra as principais funcionalidades que o sistema deverá contemplar;
- **Descrição de arquitetura**: são descritos os componentes e conectores utilizados na arquitetura proposta para que exista um entendimento geral entre os envolvidos no processo.
- **Avaliação de cada cenário**: para cada cenário é determinado se as arquiteturas propostas oferecem suporte ao cenário ou se é necessária alguma alteração.
- **Determinação da interação de cenários**: "uma interação entre cenários ocorre quando dois ou mais cenários indiretos exigem modificações em algum componente ou conexão" (MENDES, 2002, p.68). As interações entre os cenários mostram quais componentes estão associados a quais funcionalidades.
- **Avaliação de cenários e da interação de cenários**: "esta é uma etapa subjetiva do processo de análise, envolvendo os principais influenciadores do sistema. [...] Essa avaliação reflete a influência dos atributos de qualidade associados aos cenários" (MENDES, 2002, p.68).

2.3.5.3. ATAM (*Architecture Tradeoff Analysis Method*)

ATAM (*Architecture Tradeoff Analysis Method*) é um método de análise arquitetural utilizado, principalmente, para identificar riscos. Ou seja, identificar decisões para as quais não são conhecidas todas as conseqüências.

Esse é um método arquitetural que visa "compreender as conseqüências de decisões arquiteturais em relação aos requisitos dos atributos de qualidade de um sistema" (KASMAN, 1999 *apud* MENDES, 2002, p. 72). ATAM identifica, além de riscos, pontos de susceptibilidades e pontos de compromisso. ATAM não é apenas um método de análise arquitetural é, também, um método para atenuação riscos.

Os pontos de susceptibilidade são os componentes necessários para alcançar algum tipo de atributo de qualidade. Um ponto de compromisso é um ponto de susceptibilidade necessário para obtenção de múltiplos atributos.

Mendes (2002), afirma que ATAM possui três aspectos fundamentais sobre os quais o método foi desenvolvido:

- Caracterização dos atributos de qualidade do sistema de software;
- Elicitação e análise de cenários;
- Análise de decisões arquiteturais utilizando mecanismo de raciocínio denominado ABAS[6];

Para atender aos requisitos não-funcionais, ATAM procura identificar metas de negócios de uma empresa. Este método compreende nove passos distribuídos em quatro etapas:

1 ª Etapa – Apresentação

1. Apresentação do método ATAM;
2. Apresentação de metas da empresa;
3. Apresentação da arquitetura;

2 ª Etapa – Investigação e Análise

4. Identificação de abordagens arquiteturais;
5. Geração de árvore de utilidade de atributos de qualidade;
6. Análise das abordagens arquiteturais

3 ª Etapa – Teste

7. Análise, discussão e determinação de prioridade de cenários;
8. Análise das abordagens;

4 ª Etapa – Relato

9. Apresentação dos resultados.

Os nove passos deste método nem sempre são executados seqüencialmente. Às vezes é necessário voltar ou avançar algum passo para fazer interações entre passos. A seqüência em que os passos serão executados depende das necessidades do sistema.

2.3.6. Projeto arquitetural

Durante o projeto arquitetural são construídos modelos que representam a estrutura do sistema em questão. Este modelo engloba os componentes e os relacionamentos existentes entre os componentes necessários para a construção de um sistema.

Quando o projeto arquitetural se inicia, é necessário colocar o software a ser construído em contexto. Ou seja, é preciso definir com quais entidades externas o sistema irá interagir e como será essa interação. Após isso, é especificada a estrutura do sistema através

[6] *Attribute-based Architecture Style*. Ferramenta utilizada para compreender qualitativa e quantitativamente os atributos de qualidade de um estilo arquitetural.

da definição dos componentes que implementam a arquitetura. Para modelar o contexto arquitetural é possível utilizar um diagrama de contexto arquitetural (*Architectural Context Diagram* – ACD). Esse diagrama modela a forma como o sistema interagirá com entidades externas.

A Figura 11 demonstra um diagrama de contexto arquitetural.

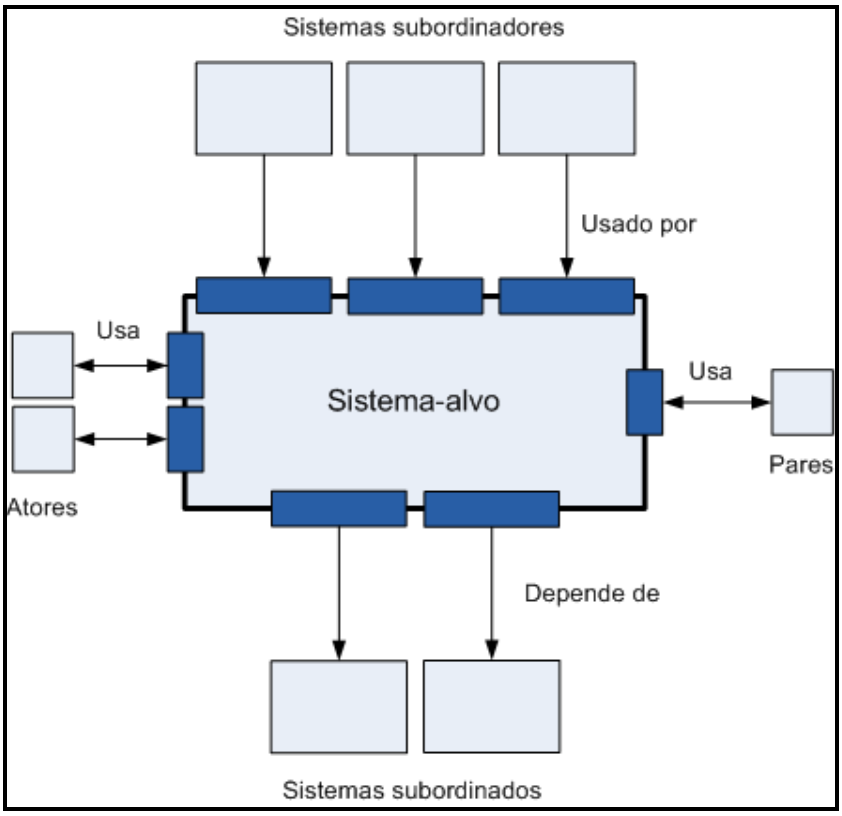

Figura 11 – Diagrama de contexto arquitetural
Fonte: Pressman (2006)

Os sistemas que interagem com sistema-alvo, àquele que está sendo desenvolvido, são representados por:

- **Sistemas subordinadores**: utilizam o sistema alvo como parte de algum tipo de processamento;
- **Sistemas subordinados**: são usados pelo sistema alvo, fornecendo dados ou processamento para o sistema alvo realizar algum tipo de função;
- **Sistemas no nível de pares**: "sistemas que interagem em base par-a-par (a informação é produzida ou consumida entre os pares e o sistema-alvo)" (PRESSMAN, 2006, p.217);
- **Atores**: pessoas ou dispositivos que interagem com o sistema alvo, produzindo ou consumindo informações.

A comunicação entre essas entidades e o sistema-alvo é realizada através de interfaces. Na Figura 11, essas interfaces são representadas por retângulos menores.

A arquitetura de software pode ser descrita através de linguagens de descrição arquitetural (*Architectural Description Language* – ADL). Essas linguagens disponibilizam a sintaxe e a semântica necessária para descrever uma arquitetura de software.

2.4. Componentes de software

Um componente de software é uma unidade de software desenvolvida para fins específicos e amplamente testado. Esses componentes devem ser úteis, adaptáveis, portáveis e, principalmente, reutilizáveis. O objetivo principal dos componentes de software é reutilizar código escrito em qualquer linguagem e que possa ser executado em qualquer plataforma.

A fase de projeto, na maior parte das disciplinas de engenharia, tem como base a reutilização de componentes. Por exemplo, na engenharia mecânica ou elétrica, os engenheiros não definem projetos onde todos os componentes terão que ser construídos especialmente para àquele projeto. Ao invés disso, eles fazem uso de componentes já desenvolvidos e testados.

Sommerville (2003, p. 260), afirma que:

> O reuso de software deve ser considerado durante o projeto de engenharia ou de requisitos. O reuso oportuno é possível durante a programação, quando se descobrem componentes que por acaso atendam a um requisito. Contudo, o reuso sistemático requer um processo de projeto que considere como os projetos existentes podem ser reutilizados e que organize explicitamente o projeto em torno de componentes de software disponíveis.

Um componente de software pode ser definido a partir de contextos diferentes:

- **Orientado a objetos**: um componente contém um conjunto de classes colaborativas. Essas classes possuem os atributos e operações necessários à sua implementação. A comunicação entre componentes ocorre através de trocas de mensagens utilizando interfaces bem definidas.
- **Convencional**: componentes são elementos funcionais de programa que incorporam "lógica de processamento, estruturas de dados internas que são necessárias para implementar a lógica de processamento e uma interfaces que possibilita ao componente ser chamado e dados serem passados por ele" (PRESSMAN, 2006, p. 241).

- **Relacionado a processo**: nesta abordagem, ao contrário das duas anteriores, os componentes não são desenvolvidos do zero. São utilizados componentes ou padrões de projeto disponibilizados em um catálogo. Esse catálogo apresenta soluções reutilizáveis e descreve todas as funcionalidades e interfaces disponíveis.

A vantagem mais óbvia da reutilização de software é a redução de custos exigidos pelo desenvolvimento de software. Além disso, existem outros benefícios decorrentes da reutilização de software. Sommerville (2003) cita alguns desses benefícios:

- **Maior confiabilidade**: os componentes reutilizáveis devem ser mais confiáveis do que componentes novos, pois eles já foram utilizados e testados em diferentes ambientes. Assim, qualquer defeito do componente já deveria ter sido descoberto e solucionado, logo o inicio da utilização do componente.
- **Redução de riscos do processo**: ao reutilizar componentes, são reduzidas as incertezas relacionadas ao custo de reutilização em relação ao custo de desenvolvimento.
- **Uso efetivo de especialistas**: ao invés dos especialistas desenvolverem os mesmos componentes em diversos projetos, eles podem desenvolver componentes reutilizáveis que englobam seu conhecimento.
- **Conformidade com padrões**: padrões, como o de interface com o usuário, podem ser construídos utilizando componentes-padrões, o que tornará a interface mais confiável, uma vez que todas as aplicações utilizarão o mesmo padrão.
- **Desenvolvimento acelerado**: "de modo geral, é mais importante fornecer um sistema para o mercado o mais rápido possível do que se prender aos custos gerais de desenvolvimento" (SOMMERVILLE, 2003, p. 261). A reutilização de componentes acelera a produção, pois, reduz o tempo de desenvolvimento e validação necessários.

Embora os componentes reutilizáveis possam trazer diversos benefícios, Sommerville (2003) identifica alguns pontos que podem acarretar problemas:

- **Aumento nos custos de manutenção**: devido às mudanças que ocorrem no sistema, os componentes podem se tornar incompatíveis. Em casos onde o código-fonte do componente não está disponível, os custos com manutenção podem aumentar.
- **Falta de ferramentas de apoio**: as ferramentas CASE[7] disponíveis no mercado não oferecem suporte ao desenvolvimento com reuso. Por isso, pode se tornar muito difícil integrar essas ferramentas com as bibliotecas de componentes.

[7] *Computer Aided Software Engineering*. São ferramentas que dão suporte as atividades de desenvolvimento de software.

- **Síndrome do "não-foi-inventado-aqui"**: alguns desenvolvedores preferem reescrever componentes ao invés de reutilizar os existentes. Isso ocorre porque esses desenvolvedores acreditam que podem fazer um componente melhor do que os disponíveis. "Isso tem a ver em parte com confiança e em parte com o fato de que escrever um software original é visto como mais desafiador do que reutilizar o software de outras pessoas" (SOMMERVILLE, 2003, p.262).
- **Manutenção de uma biblioteca de componentes**: as técnicas para catalogação de componentes reutilizáveis são muito imaturas, dessa forma, é difícil garantir que os desenvolvedores de software vão utilizar as bibliotecas desenvolvidas.
- **Encontrar e adaptar componentes reutilizáveis**: é preciso ter certeza de que um componente será encontrado em uma biblioteca, além disso, é necessário compreender e, às vezes, adaptar os componentes encontrados.

Decompor um sistema em componentes reutilizáveis resulta no desenvolvimento de interfaces que servem como conectores entre os componentes e entre componentes e a aplicação.

2.4.1. Engenharia de software baseada em componentes

Atualmente, sistemas complexos precisam ser construídos em pouco tempo, consumindo poucos recursos e apresentando alta qualidade. Para alcançar estes objetivos, a engenharia de software baseada em componentes (*Componente-Based Software Engineering* – CBSE), também conhecida como desenvolvimento baseado em componentes, enfatiza a construção de software utilizando componentes reutilizáveis.

O desenvolvimento baseado em componentes teve como principal motivação:

> A frustração de que o desenvolvimento orientado a objetos não tinha conduzido a um extensivo reuso, como originalmente foi sugerido. As classes de objetos individuais eram muito detalhadas e específicas e tinham de estar associadas a uma aplicação em tempo de compilação ou quando o sistema estivesse conectado. O conhecimento detalhado das classes era necessário para sua utilização, e isso, geralmente, significa que o código fonte precisava estar disponível, apresentando problemas difíceis para a comercialização de componentes. Apesar das primeiras previsões otimistas, nenhum mercado significativo para os objetos individuais foi desenvolvido. (SOMMERVILLE, 2003, p. 263).

Componentes são mais abstratos do que classes e são considerados como provedores de serviços. Ou seja, quando um sistema precisar de algum serviço, ele acessará um

componente para fornecer este serviço. Para o sistema, não importa onde o componente está sendo executado ou em que linguagem de programação foi desenvolvido.

Os componentes de software podem apresentar abstrações em diferentes níveis. De acordo com Sommerville (2003), algumas dessas abstrações são:

- **Abstração funcional**: os componentes implementam apenas uma função, como uma função matemática, por exemplo;
- **Agrupamentos casuais**: os componentes são conjuntos de entidades inadequadamente relacionadas;
- **Abstrações de dados**: os componentes representam abstrações de dados ou classes em uma linguagem orientada a objetos;
- **Abstrações de clusters**: são conjuntos de classes relacionadas que trabalham em conjunto, às vezes, são chamados de *frameworks*;
- **Abstração de sistema**: "o componente é um sistema inteiramente autocontido. Reutilizar abstrações de nível de sistema é, às vezes, chamado de reuso de produtos COTS" (SOMMERVILLE, 2003, p. 264) Os *Commercial off-the-shelf* (COTS), são produtos de prateleira, fornecidos por terceiros (fabricantes), que oferecem funcionalidades mais amplas que os sistemas especializados desenvolvidos para uma organização específica.

Um processo de CBSE, além de identificar candidatos a componentes, qualifica a interface dos componentes, adapta os componentes removendo discordâncias arquiteturais, monta componentes em um estilo arquitetural selecionado e atualiza esses componentes à medida que os requisitos do sistema se modificam.

2.4.2. Famílias de aplicação (ou linha de produtos)

Famílias de aplicação, ou linha de produtos, são conjuntos de aplicações relacionadas que possuem uma arquitetura de domínio específico comum. O núcleo comum à família de aplicações é reutilizado sempre que for necessário desenvolver algum componente adicional ou adaptar componentes existentes.

A especialização de uma família de aplicações pode ocorrer de diversas maneiras, Sommerville (2003) cita algumas:

- **Especialização de plataforma**: versões diferentes da mesma aplicação são desenvolvidas para serem executadas em diferentes plataformas;

- **Especialização da configuração**: versões diferentes da mesma aplicação são desenvolvidas para serem executadas em dispositivos periféricos diferentes;
- **Especialização funcional**: versões diferentes da mesma aplicação são desenvolvidas para atender requisitos de diferentes clientes.

2.4.3. Projeto de componentes baseados em classe

Ao escolher uma abordagem orientada a objetos, o projeto no nível de componentes passa a ser direcionado à construção de classes específicas do domínio do problema. Existem alguns princípios de projetos que podem ser seguidos ao utilizar a engenharia de software orientada a objetos. Esses princípios têm como objetivo aumentar a manutenibilidade do projeto e diminuir o impacto de modificações.

Martin (2000 *apud* PRESSMAN, 2006), define alguns desses princípios:

- **Princípio Aberto-Fechado** (*Open-Closed Principle – OCP*): um módulo (componente) deveria ser aberto para extensão, mas fechado para modificação. Ou seja, o componente deve fornecer a capacidade de ser estendido, mas sem a necessidade de modificações internas no componente.
- **Princípio de Substituição de Liskov** (*Liskov Substitution Principle – LSP*): subclasses devem se substituíveis por suas classes-base. Componentes que utilizam uma classe-base devem continuar funcionando se receberem uma classe derivada da classe-base.
- **Princípio da Inversão de Dependência** (*Dependency Inversion Principle – DIP*): "Confie nas abstrações. Não confie nas concretizações" (MARTIN, 2000 *apud* PRESSMAN, 2006, p. 245). Quanto maior for à dependência de um componente de outro componente concreto, mais difícil será estendê-lo.
- **Princípio da Segregação de Interface** (*Interface Segregation Principle – ISP*): "Muitas interfaces específicas de clientes são melhores do que uma interface de propósito geral" (MARTIN, 2000 *apud* PRESSMAN, 2006, p. 245). Este princípio sugere a criação de interfaces especializadas para atender a cada tipo de cliente.
- **Princípio de Equivalência de Liberação de Reuso** (*Release Reuse Equivalency Principle – REP*): "A granularidade do reuso é a granularidade da versão" (MARTIN, 2000 *apud* PRESSMAN, 2006, p. 245). Deve existir compatibilidade entre as versões

do sistema. É necessário que as versões mais recentes do componente forneçam suporte às versões mais antigas para que as atualizações ocorram gradativamente.

- **Princípio de Fecho Comum** (*Common Closure Principle – CCP*): "Classes que se modificam juntas devem ficar juntas" (MARTIN, 2000 *apud* PRESSMAN, 2006, p. 245). Classes que tratam de um determinado aspecto do sistema devem ser mantidas juntas. Com isso, é possível ter um controle maior sobre as modificações que possam ocorrer.
- **Princípio Comum de Reuso** (*Common Reuse Principle – CRP*): "Classes que não são reusadas juntas não devem ser agrupadas juntas" (MARTIN, 2000 *apud* PRESSMAN, 2006, p. 246). Classes que não possuem finalidades em comum devem ser mantidas separadas.

2.5. Orientação a objetos

Durante a década de 1960, os computadores contavam com capacidade de processamento muito reduzida. Isso obrigava os desenvolvedores a construir sistemas simples e pequenos, que aproveitassem o máximo da memória e do processamento da máquina. Nessa época não existiam metodologias de desenvolvimento e "muitos acreditavam que, uma vez que os algoritmos atingissem um ponto ideal, não seria mais necessário alterar ou efetuar manutenção dos programas até que deixassem de ser usados" (LIMA, 2005, p.17).

Na década de 1970, a capacidade de processamento dos computadores já havia aumentado. Nessa época começaram a surgir padrões de desenvolvimento modularizados. As funções passaram a ser o ponto principal dos sistemas. No entanto, apesar de ser mais fácil de compreender e pudessem ser reutilizadas em outros sistemas, "as funções não contemplavam a administração de dados e suas mudanças aumentavam a necessidade de manutenção dos programas" (LIMA, 2005, p.18). Isso ocorria porque os desenvolvedores se preocupavam mais com os procedimentos do que com os dados.

Quando a importância dos bancos de dados relacionais começou a aumentar, o desenvolvimento "passou a centrar-se nos dados e nos métodos de modelagem no modelo entidade-relacionamento (MER), pois entendia-se que os dados eram a parte mais estável de um aplicação" (LIMA, 2005, p.18) Porém, as funções passaram a ser ignoradas.

A Orientação a Objetos (OO) não é uma técnica nova. Na década de 1960, surgiram as primeiras linguagens de programação, como Simula e Smalltalk, que sustentavam "o princípio

de que os programas deviam ser estruturados com base no problema a se resolvido, fundamentando a construção nos objetos de negócio em si" (LIMA, 2005, p. 18).

A orientação a objetos é uma técnica utilizada para modelar, analisar, projetar e desenvolver software em termos de objetos. Ou seja, a orientação a objetos utiliza um conjunto de entidades que possuem características e comportamentos próprios, tais como objetos do mundo real, como telefones, produtos, pessoas, etc. Blaha e Rumbaugh (2006, p. 1) definem a modelagem e projeto orientado a objetos como:

> Um modo de pensar a respeito dos problemas aplicando-se modelos organizados em torno de conceitos do mundo real. A construção fundamental é o objeto, que combina estrutura de dados e comportamento. Os modelos orientados a objetos são úteis para entender problemas, comunicar-se com especialistas de aplicação, modelar empresas, preparar documentação e projetar programas e bancos de dados.

Existe muita discordância sobre as características exatas da orientação a objetos. No entanto, Blaha e Rumbaugh (2006), citam quatro aspectos principais da orientação a objetos: objetos, classes, herança e polimorfismo.

Os objetos são entidades distintas que possuem características e comportamentos próprios. Um objeto é "um conceito, abstração ou coisa com identidade que possui significado para a aplicação" (BLAHA e RUMBAUGH, 2006, p. 23).

As classes são conjuntos de objetos que possuem estruturas de dados (atributos) e comportamento (operações) em comum. Uma classe "representa uma idéia ou um conceito simples e categoriza objetos que possuem propriedades similares, configurando-se em um modelo para a criação de novas instâncias" (FURLAN, 1998, p.17). Ou seja, uma classe descreve grupos de objetos com características em comum. Uma classe define a estrutura de um objeto sem definir seus valores, enquanto um objeto é uma instância de uma classe em um determinado momento.

Herança é o compartilhamento de atributos e métodos entre classes. Esse compartilhamento ocorre através da definição de subclasses que herdam características de uma superclasse através de um relacionamento hierárquico.

Polimorfismo significa que uma mesma operação pode se comportar de maneiras diferentes. Segundo Melo (2002, p. 22) "uma operação pode ter implementações diferentes em diversos pontos da hierarquia de classes, desde que mantenham a mesma assinatura. As operações das subclasses prevalecerão sobre as da superclasse".

Além dos princípios citados acima, existe uma série de outros conceitos relacionados à orientação a objetos:

- **Classes abstratas**: representam entidades ou conceitos abstratos. Não podem ser instanciadas diretamente, sendo necessária a existência de subclasses que serão instanciadas. É sempre uma superclasse.
- **Interfaces**: os métodos declarados em um objeto possuem nome, parâmetros e valores de retorno, isso é conhecido como assinatura da operação. Uma interface representa o conjunto de assinaturas que um objeto possui. A interface de um objeto representa "o conjunto completo de solicitações que podem ser enviadas para o mesmo. Qualquer solicitação que corresponde a uma assinatura na interface do objeto pode ser enviada para o mesmo" (GAMMA *et al*, 2000, p. 29).
- **Mensagens**: é a forma como os objetos se comunicam, é a maneira como um objeto solicita um serviço através de uma operação.
- **Composição**: é uma alternativa à herança de classe. Com a composição, novas funcionalidades são obtidas através da montagem de objetos. "A composição de objetos é definida dinamicamente em tempo de execução pela obtenção de referências para outros objetos por um determinado objeto" (GAMMA *et al*, 2000, p. 34).
- **Delegação**: permite tornar a composição tão poderosa quanto à herança. Durante a delegação, "dois objetos são envolvidos no tratamento de uma solicitação: um objeto receptor delega operações para o seu delegado; isto é análogo à postergação de solicitações enviadas às subclasses para suas classes-mãe" (GAMMA *et al*, 2000, p. 35).
- **Abstração**: permite se concentrar em aspectos importantes, ignorando os detalhes. A abstração possibilita entender o que um objeto é e faz, antes de implementá-lo.
- **Encapsulamento**: "separa os aspectos externos de um objeto, que são acessíveis a outros objetos, dos detalhes internos da implementação, que estão escondidos de outros objetos" (BLAHA e RUMBAUGH, 2006, p. 7).
- **Tipos parametrizados**: "permite definir um tipo sem especificar todos os outros tipos que ele usa. Os tipos não-especificados são fornecidos como parâmetros no ponto de utilização" (GAMMA *et al*. 2000, p. 37). Por exemplo, em *Java*, um objeto da classe *List* pode ser parametrizado para aceitar apenas valores inteiros.

Uma das vantagens da orientação a objetos é a unificação entre dados e processos. De acordo com Correia e Tafner (2001, p.105),

> Uma antiga questão de análise de sistema é a consistência entre a definição dos dados e dos processos. Algumas metodologias recomendam definir a estrutura de dados primeiro, e em seguida definir os processos que os utilizam. Outras

> recomendam o oposto: procurar definir os procedimentos que o sistema automatiza e posteriormente identificar os dados que esses processos necessitam.

Na orientação a objetos esse problema não existe, pois, os dados e os processos são apenas componentes. O objetivo principal é identificar como os objetos interagem entre si. Outra vantagem da orientação a objetos é a consistência entre análise e desenvolvimento. Em metodologias não orientadas a objetos, normalmente é utilizado o Modelo Entidade Relacionamento (MER) e o Diagrama de Fluxo de Dados (DFD) para separar os dados dos processos que manipulam esses dados. Enquanto na orientação a objetos as classes e objetos definidos durante a análise são os mesmos que serão implementados.

A orientação a objetos também torna mais fácil a manutenção dos sistemas. As modificações necessárias são realizadas apenas nas classes que precisam ser alteradas e, essas alterações são refletidas no sistema inteiro. Ao contrário da programação convencional, onde:

> A alteração de uma funcionalidade que é utilizada em vários lugares do sistema pode se tornar um pesadelo. O programador que efetuar a manutenção pode não conseguir localizar todos os trechos de código que acessam a funcionalidade, e algumas vezes ao realizar a alteração, são incluídos erros em alguns dos locais alterados (CORREIA e TAFNER, 2001, p.105).

2.6. Padrões de projeto

O conceito de utilização de padrões de projeto (*design patterns*) para construir sistemas de softwares originou-se no campo da construção civil. Mais especificamente na área da arquitetura, onde os arquitetos utilizam elementos de projetos arquitetônicos, como arcos e colunas, para projetar edifícios. No caso dos sistemas de software, esses elementos são conjuntos de classes e objetos. *Christopher Alexander* foi um dos primeiros a propor a idéia de utilizar uma linguagem de padrões em projetos de edificações e cidades.

Entre 1991 e 1994, *Erich Gamma*, *Richard Helm*, *Ralph Johnson* e *John Vlissides*, também conhecidos como "*Gang of Four*" ou "*GoF*", escreveram o livro *Design patterns: elements of reusable object-oriented software*. Esse livro, ou catálogo, descreve 23 padrões de projeto. Cada um desses padrões fornece uma solução a um problema comum de projeto de software.

Após a publicação do livro *Design Patterns*, diversos padrões de projeto adicionais foram criados, no entanto, o catálogo da *Gang of Four* continua sendo uma referência importante quando se trata de padrões de projeto.

Para Gamma *et al.* (2000), toda a arquitetura orientada a objetos bem-estruturada possui diversos padrões. Esses autores afirmam, também, que uma das maneiras de avaliar a qualidade de um sistema orientado a objetos, é avaliar se os desenvolvedores tomaram bastante cuidado com as colaborações entre os objetos. Tratar este aspecto durante o desenvolvimento de um sistema pode levar a uma arquitetura menor, mais simples e compreensível que as produzidas quando estes padrões são ignorados.

Padrões de projeto tornam mais fácil a reutilização de projetos e arquiteturas bem-sucedidas, pois ajudam a escolher alternativas de projeto que tornam um sistema reutilizável e a evitar alternativas que comprometam a reutilização. Além disso, ao fornecer uma especificação explícita de interações de classe e objetos e o seu objetivo, os padrões de projeto podem melhorar a documentação e a manutenção de sistemas (GAMMA *et al.*, 2000).

2.6.1. O que são padrões de projeto

Os padrões de projeto capturam soluções que foram desenvolvidas e aperfeiçoadas ao longo do tempo. Esses padrões refletem a experiência a os esforços dos desenvolvedores por maior reutilização e flexibilidade.

Gamma *et al*. (2000, p.324) afirmam que os "padrões de projeto fornecem um vocabulário comum para comunicar, documentar e explorar alternativas de projeto". Esses padrões tornam um sistema menos complexo, uma vez que permitem falar sobre o sistema em um nível de abstração mais alto do que aquele fornecido por linguagens de programação.

De acordo com Gamma *et al.* (2000, p.20):

> Um padrão de projeto nomeia, abstrai e identifica aspectos-chave de uma estrutura de projeto comum para torná-la útil para a criação de um projeto orientado a objetos reutilizável. O padrão de projeto identifica as classes e instâncias participantes, seus papéis, colaborações e distribuição de responsabilidades. Cada padrão de projeto focaliza um problema ou tópico particular de projeto orientado a objetos. Ele descreve quando pode ser aplicado, se ele pode ser aplicado em função de outras restrições de projeto e as conseqüências, custos e benefícios de sua utilização.

Deitel (2005, p.1) define padrões de projeto como "arquiteturas testadas para construir softwares orientados a objetos flexíveis e sustentáveis". Este autor também afirma que padrões de projeto ajudam a reduzir a complexidade do processo de *design*.

Os padrões de projeto não exigem nenhum recurso incomum, nem truques das linguagens de programação. Esses padrões podem ser implementados em linguagem orientada a objetos comum. Alguns podem até mesmo ser implementados em linguagens procedurais,

embora possam exigir um pouco mais de trabalho. No entanto, esse esforço adicional trará como resultado maior flexibilidade.

Em geral, um padrão de projeto é composto por quatro elementos:

1. **Nome**: é a referência utilizada para descrever um problema de projeto, suas soluções e conseqüências em uma ou duas palavras.
2. **Problema**: descreve quando o padrão pode, ou deve, ser utilizado. Pode descrever problemas de projeto específicos, estruturas de classes ou objetos de um projeto inflexível. O problema pode incluir uma lista de condições que devem ser satisfeitas para que faça sentido aplicar o padrão.
3. **Solução**: descreve os elementos que compõem o projeto, seus relacionamentos, suas responsabilidades e colaborações. A solução não descreve um projeto concreto ou uma implementação.
4. **Conseqüências**: são os resultados e análises das vantagens e desvantagens da aplicação do padrão de projeto.

2.6.2. Classificação dos padrões de projeto

Os padrões de projeto do catálogo da *Gang of Four* podem ser classificados seguindo dois critérios. O primeiro critério refere-se à finalidade, que reflete o que um padrão faz. O segundo critério diz respeito ao escopo, e especifica se o padrão se aplica a classes ou objetos.

A Tabela 5 demonstra a classificação dos padrões de projeto.

Tabela 5 – Classificação dos padrões de projeto

		Propósito		
		De criação	**Estrutural**	**Comportamental**
Escopo	**Classe**	*Factory Method*	*Adapter (class)*	*Interpreter* *Template Method*
	Objeto	*Abstract Factory* *Builder* *Prototype* *Singleton*	*Adapter (object)* *Bridge* *Composite* *Decorator* *Façade* *Flyweight* *Proxy*	*Chain of responsibility* *Command* *Iterator* *Mediator* *Memento* *Observer* *State* *Strategy* *Visitor*

Fonte: Gamma *et al.* (2000)

2.6.2.1. Padrões de criação

Os padrões de criação tornam um sistema independente da forma como este cria, compõem e representa seus objetos. “Um padrão de criação de classe usa herança para variar a classe que é instanciada, enquanto que um padrão de criação de objetos delegará a instanciação para outro objeto” (GAMMA *et al*. *2000, p.92).*

A Tabela 6 apresenta a descrição resumida dos padrões de projeto de criação descritos no catálogo da *Gang of Four*.

Tabela 6 – Padrões de projeto de criação

Padrão	**Descrição**
Abstract Factory	Fornece uma interface para criação de famílias de objetos relacionados ou dependentes sem especificar suas classes concretas.
Builder	Separa a construção de um objeto complexo da sua representação, de modo que o mesmo processo de construção possa criar diferentes representações.
Factory Method	Define uma interface para criar um objeto, mas deixa as subclasses decidirem qual será instanciada. O *Factory Method* permite uma classe postergar (*defer*) a instanciação a subclasses.
Prototype	Especifica os tipos de objetos a serem criados usando uma instância prototípica e cria novos objetos copiando este protótipo.
Singleton	Garante que uma classe tenha somente uma instância e fornece um ponto global de acesso para ela.

Fonte: Gamma *et al.* (2000)

2.6.2.2. Padrões estruturais

Padrões estruturais preocupam-se com a forma como classes e objetos são compostos para formar uma estrutura maior. Esses padrões podem utilizar herança para compor interfaces ou implementações, ou descrever maneiras de compor objetos para obter novas funcionalidades (GAMMA *et al*., 2000).

A Tabela 7 apresenta a descrição resumida dos padrões estruturais descritos no catálogo da *Gang of Four*.

Tabela 7 – Padrões de projeto estruturais

Padrão	**Descrição**
Adapter	Converte a interface de uma classe em outra interface esperada pelos clientes. O *Adapter* permite que certas classes trabalhem em conjunto, pois de outra forma seria impossível por causa de suas interfaces incompatíveis.
Bridge	Separa uma abstração da sua implementação, de modo que as duas possam variar independentemente.
Composite	Compõe objetos em estruturas de árvore para representar hierarquias do tipo parte-todo. O *Composite* permite que os clientes tratem objetos individuais e composições de objetos de maneira uniforme.
Decorator	Atribui responsabilidades adicionais a um objeto dinamicamente. Os *decorators* fornecem uma alternativa flexível a subclasses para extensão da funcionalidade.
Façade	Fornece uma interface unificada para um conjunto de interfaces em um subsistema. O *Façade* define uma interface de nível mais alto que torna o subsistema mais fácil de usar.
Flyweight	Usa compartilhamento para suportar grandes quantidades de objetos, de granularidade fina, de maneira eficiente.
Proxy	Fornece um objeto representante (*surrogate*), ou um marcador de outro objeto, para controlar o acesso ao mesmo.

Fonte: Gamma *et al.* (2000)

2.6.2.3. Padrões comportamentais

Os padrões comportamentais preocupam-se com algoritmos e atribuição de responsabilidades entre os objetos. Não descrevem apenas padrões de objetos ou classes, mas também padrões de comunicação entre eles. Estes padrões caracterizam fluxos de controle difíceis de seguir em tempo de execução (GAMMA *et al.*, 2000).

A Tabela 8 apresenta a descrição resumida dos padrões comportamentais descritos no catálogo da *Gang of Four*.

Tabela 8 – Padrões de projeto comportamentais

Padrão	**Descrição**
Chain of Responsibility	Evita o acoplamento do remetente de uma solicitação ao seu destinatário, dando a mais de um objeto a chance de tratar a

	solicitação. Encadeia os objetos receptores e passa a solicitação ao longo da cadeia até que um objeto a trate.
Command	Encapsula uma solicitação como um objeto, desta forma permitindo parametrizar clientes com diferentes solicitações, enfileirar ou registrar (*log*) solicitações e suportar operações que podem ser desfeitas.
Interpreter	Dada uma linguagem, define uma representação para sua gramática juntamente com um interpretador que usa a representação para interpretar sentenças nessa linguagem.
Iterator	Fornece uma maneira de acessar seqüencialmente os elementos de um objeto agregado sem expor sua representação subjacente.
Mediator	Define um objeto que encapsula como um conjunto de objetos interage. O *Mediator* promove o acoplamento fraco ao evitar que os objetos se refiram explicitamente uns aos outros, permitindo variar suas interações independentemente.
Memento	Sem violar a encapsulação, captura e externaliza um estado interno de um objeto, de modo que o mesmo possa posteriormente ser restaurado para este estado.
Observer	Define uma dependência um-para-muitos entre objetos, de modo que, quando um objeto muda de estado, todos os seus dependentes são automaticamente notificados e atualizados.
State	Permite que um objeto altere seu comportamento quando seu estado interno muda. O objeto parecerá ter mudado sua classe.
Strategy	Define uma família de algoritmos, encapsula cada um deles e torna-os intercambiáveis. O *Strategy* permite que o algoritmo varie independentemente dos clientes que o utilizam.
Template Method	Define o esqueleto de um algoritmo em uma operação, postergando a definição de alguns passos para subclasses. O *Template Method* permite que as subclasses redefinam certos passos de um algoritmo sem mudar sua estrutura.
Visitor	Representa uma operação a ser executada sobre os elementos da estrutura de um objeto. O *Visitor* permite definir uma nova

	operação sem mudar as classes dos elementos sobre os quais opera.

Fonte: Gamma *et al.* (2000)

2.6.3. Padrões GRASP (*General Responsibility Assignment Software Patterns*)

Os Padrões de Princípios Gerais para Atribuição de Responsabilidades (*General Responsibility Assignment Software Patterns* – GRASP) descrevem princípios para a atribuição de responsabilidades a objetos. Esses padrões não propõem nenhuma idéia nova, eles apenas codificam princípios que já são amplamente conhecidos. A Tabela 9 explica rapidamente cada um dos nove padrões GRASP.

Tabela 9 – Padrões de projeto GRASP

Padrão	**Descrição**
Expert	Atribui responsabilidades a classes que possui a informação necessária para atender determinada responsabilidade.
Creator	Atribui a uma classe B a responsabilidade de instanciar classes A se uma das seguintes condições for verdadeira (LARMAN, 2000): ▪ B agrega objetos A; ▪ B contém objetos A; ▪ B registra instâncias de objetos A; ▪ B usa de maneira muito próxima objetos A; ▪ B tem os dados de inicialização que serão passados para objetos A, durante a criação de A.
Low Coupling	Atribuem responsabilidades mantendo o acoplamento fraco entre as classes. O acoplamento diz respeito ao quão interligadas estão às classes de um sistema.
High Cohesion	Atribui responsabilidades mantendo a alta coesão entre as classes. Classes têm coesão alta quando possuem responsabilidades altamente relacionadas, ou seja, só executa tarefas relacionadas a ela.
Controller	Atribui responsabilidades do tratamento de uma mensagem de um evento do sistema a uma classe que represente uma das seguintes

	opções (LARMAN, 2000): ▪ Representa o "sistema" todo (controlador fachada); ▪ Representa todo o negócio ou organização (controlador fachada); ▪ Representa algo no mundo real que é ativo e pode estar envolvido na tarefa (controlador do papel); ▪ Representa um tratador artificial de todos os eventos de sistema de um caso de uso, geralmente chamado controlador do caso de uso (controlador do caso de uso).
Polymorphism	Atribui responsabilidades de acordo com o comportamento, que pode variar de acordo com a classe, utilizando operações polimórficas.
Pure Fabrication	Atribui responsabilidades a classes que não representam nada no domínio do problema. Essas classes são construídas apenas para manter a coesão alta e o acoplamento fraco.
Indirection	Atribui responsabilidades a objetos intermediários. Esses objetos estão entre outros componentes, evitando que estes sejam altamente acoplados.
Don't Talk to Stangers	Atribui responsabilidades a objetos diretamente referenciados por clientes. Procura evitar a visibilidade temporária de objetos que são conhecidos por outros objetos, mas não pelos clientes.

Fonte: Kamyla Estuqui Parrado (2008)

2.6.4. Diferenças entre padrões de projeto e *frameworks*

Devido às semelhanças existentes entre padrões de projeto e *frameworks*, muitas vezes surgem dúvidas em relação às diferenças existentes entre os dois. Gamma *et al.* (2000) citam três pontos que diferenciam padrões de projeto de *frameworks*:

1. **Padrões de projeto são mais abstratos que *frameworks***, ou seja, os *frameworks* podem ser representados através de código, porém, apenas exemplos de padrões de projeto podem ser representados dessa forma.
2. **Padrões de projeto são elementos da arquitetura menores que *frameworks***, pois, um *framework* normalmente contém diversos padrões de projeto, porém a o oposto não é possível.

3. **Padrões de projeto são menos especializados que *frameworks***. Os *frameworks* sempre atendem a um domínio específico, enquanto os padrões de projeto podem ser aplicados a quase qualquer tipo de domínio.

A Tabela 10 demonstra algumas diferenças entre os padrões e *frameworks* apontadas por Kuchana (2004).

Tabela 10 – Padrões de projeto *versus frameworks*

Design Patterns	***Frameworks***
Design patterns são soluções repetidas para problemas que aparecem durante a vida de um software em um contexto particular.	Um *framework* é um grupo de componentes que cooperam uns com os outros para fornecer uma arquitetura reutilizável para aplicações de um dado domínio.
Patterns são lógicos por natureza.	*Frameworks* são mais físicos por natureza, eles existem em forma de software.
As descrições de *patterns* são geralmente independentes da linguagem de programação ou detalhes de implementação.	Pelo fato dos *frameworks* existirem na forma de software, eles são uma implementação específica.
Patterns são mais genéricos e podem ser usado em quase todos os tipos de aplicação.	*Frameworks* fornecem funcionalidades para um domínio específico.
Um *design pattern* não existe em forma de componente de software por conta própria. Ele precisa ser implementado explicitamente cada vez que for utilizado.	*Frameworks* não são aplicações completas por conta própria. Aplicações completas podem ser construídas pela herança de componentes.
Patterns fornecem uma boa maneira de projetar e são usados para ajudar a projetar *frameworks*.	*Design patterns* podem ser usados no projeto e implementação dos *frameworks*. Em outras palavras, *frameworks* geralmente incorporam vários *design patterns*.

Fonte: Kuchana (2004)

2.6.5. Benefícios da utilização de padrões de projeto

Quando *frameworks* são desenvolvidos utilizando padrões de projeto, isso ajuda não somente a definir uma estrutura que utiliza soluções comprovadamente eficazes, mas também auxilia a documentar o *framework*. As pessoas que conhecem os padrões de projeto passam a

ter um rápido entendimento de como o *framework* foi estruturado. E, mesmo as pessoas que não conhecem os padrões de projeto, podem se beneficiar desta documentação. Isso ocorre porque os padrões de projeto são amplamente utilizados e possuem muita documentação associada a eles. Uma vez compreendido como funcionam os padrões de projeto, entender a estruturação do *framework* pode se tornar bem mais simples.

Padrões de projeto facilitam a reutilização de projetos e arquiteturas bem-sucedidas. Segundo Gamma *et al.* (2000, p.18),

> Expressar técnicas testadas e aprovadas as torna mais acessíveis para os desenvolvedores de novos sistemas. Os padrões de projeto ajudam a escolher alternativas de projeto que tornam um sistema reutilizável e a evitar alternativas que comprometam a reutilização. Os padrões de projeto podem melhorar a documentação e a manutenção de sistemas ao fornecer uma especificação explícita de interações de classe e objetos e o seu objetivo subjacente. Em suma, ajudam um projetista a obter um projeto "certo" mais rápido.

Outros benefícios decorrentes da utilização de padrões de projeto são:

- Auxiliam na construção de softwares confiáveis que utilizam arquiteturas testadas;
- Permitem a reutilização de projetos em futuros sistemas;
- Identificam problemas comuns e armadilhas que podem ocorrer durante a construção de sistemas;
- Permitem projetar sistemas independentemente da linguagem em que serão implementados;
- Definem um vocabulário comum de projeto entre os desenvolvedores;
- Diminuem a fase de projeto no processo de desenvolvimento de software.

2.7. *Frameworks*

A reutilização de software sempre foi um dos principais objetivos da engenharia de software. Através da reutilização de software é possível aumentar a qualidade dos produtos e diminuir o tempo de desenvolvimento. Existem diversas maneiras de reutilização de software. Algumas delas são: componentes de software, padrões de projeto, orientação a objetos e *frameworks*.

Os *frameworks* permitem a criação de famílias de aplicações. Essas aplicações são construídas a partir de uma estrutura que representa os conceitos comuns de um determinado domínio. Um *framework* define a arquitetura que as aplicações construídas a partir dele devem possuir. Um *framework* também captura as decisões de projeto referentes a um

domínio específico. Por isso, os *frameworks* não reutilizam apenas código, mas também análise e projeto.

A literatura possui muitas definições para *framework*, no entanto, nenhuma dessas definições é contrária a outra. Na verdade, cada definição complementa a outra. Algumas definições são:

- *Frameworks* representam arquiteturas desenvolvidas para obter a máxima reutilização. São compostos por um conjunto de classes abstratas e concretas, que possuem grande potencial de especialização (MATTSSON, 1996 *apud* BARRETO, 2006).
- *Framework* pode ser visto como um software parcialmente completo projetado para ser instanciado. O *framework* define uma arquitetura para uma família de subsistemas e oferece os construtores necessários para criá-los. (BUSCHMANN *et al*, 1996 *apud* BARRETO, 2006).
- *Frameworks* são compostos por objetos que colaboram entre si com o objetivo de atender a um conjunto de responsabilidades para uma aplicação específica ou um domínio de aplicação (JOHNSON, 1991 *apud* BARRETO, 2006).
- *Frameworks* são combinações de componentes que simplificam o desenvolvimento de aplicações (PETERS e PEDRYCZ, 2001).
- *Frameworks* são estruturas genéricas que podem ser ampliadas para desenvolver subsistemas ou aplicações específicas (SOMMERVILLE, 2003).
- *Frameworks* definem o comportamento de um conjunto de objetos. Eles fornecem uma maneira de reutilizar tanto projeto quanto código. Um *framework* é "um conjunto de blocos de construção de software pré-fabricados que os programadores podem usar, estender, ou customizar" (TALIGENT, 1994, p.1).
- "Um *framework* é uma estrutura de classes inter-relacionadas, que correspondem a uma implementação incompleta para um conjunto de aplicações de um domínio" (SCHWEBEL, 2005, p.31).

Segundo Schwebel (2005, p.23), a definição de *framework* mais utilizada é a de Johnson (1997 *apud* SCHWEBEL, 2005) que afirma que "um *framework* orientado a objetos é um projeto reusável como um todo ou em parte, sendo representado por um conjunto de classes abstratas e pela maneira que suas instâncias interagem".

Um *framework* não é apenas um conjunto de classes. As classes existentes em um *framework* estão inter-relacionadas e fornecem uma infra-estrutura para o desenvolvedor de aplicações. Devido a essa infra-estrutura, os *frameworks* diminuem muito a quantidade de

código que os desenvolvedores de aplicação precisam desenvolver e testar. Além disso, a infra-estrutura determina a arquitetura das aplicações construídas a partir do *framework.*

O principal objetivo dos *frameworks* é definir uma estrutura genérica em relação ao domínio tratado. Além disso, os *frameworks* precisam ser específicos o suficiente para atender ao domínio para o qual está sendo construído e, ainda ser flexível o bastante para possibilitar a construção de famílias de aplicação.

Ao projetar um *framework*, é necessário identificar quais são os conceitos comuns ao domínio tratado e quais são os conceitos específicos de cada aplicação. Essa diferenciação entre conceitos é necessária para manter no *framework* apenas as características comuns a todas as aplicações. E, também, para permitir que seja definida uma estrutura que suporte alterabilidade. A alterabilidade se refere à capacidade que o *framework* deve possuir para alterar suas funcionalidades em função de uma aplicação específica sem conseqüências imprevisíveis.

Outro ponto importante que deve ser levado em consideração ao projetar *frameworks* é a sua extensibilidade, que, no caso dos *frameworks*, se refere à manutenibilidade. A estrutura de classes de um *framework* é bem mais complexa do que a estrutura de classes das aplicações construídas a partir dele. Essa estrutura é obtida através de ciclos iterativos. A evolução dessa estrutura costuma ocorrer durante toda a vida do *framework*, seja para correção de erros, inclusão ou aprimoramento de funcionalidades. Conforme o *framework* for utilizado, novos conceitos podem surgir, por isso, é importante que o *framework* de suporte a esses novos conceitos. Segundo Silva (2000, p.45),

> Em termos práticos, dotar um *framework* de generalidade, alterabilidade e extensibilidade requer uma cuidadosa identificação das partes que devem ser mantidas flexíveis e a seleção de soluções de projeto de modo a produzir uma arquitetura bem estruturada. Isto passa pela observação de princípios de projeto orientado a objetos, como o uso de herança para reutilização de interfaces (ao invés do uso de herança para reutilização de código); reutilização de código através de composição de objetos; preocupação em promover o polimorfismo, na definição das classes e métodos, de modo a possibilitar acoplamento dinâmico etc.

Um *framework* bem projetado possui quatro características fundamentais:

- **Completo**: *frameworks* devem fornecer implementações padrão para as funcionalidades disponíveis. Dessa forma, os desenvolvedores de aplicação podem se focar apenas nos pontos que precisam ser customizados para atender a um determinado contexto.
- **Flexível**: as abstrações fornecidas podem ser usadas em diversos contextos.

- **Extensível**: os desenvolvedores de aplicação podem adicionar ou modificar o comportamento do *framework* através da extensão de classes existentes no *framework*.
- **Compreensível**: a interação entre os desenvolvedores de aplicação e o *framework* deve ser clara. Para que isso ocorra, o *framework* deve ser bem documentado e utilizar boas práticas de programação.

Segundo Gamma *et al*. (2000, p. 43),

> Os *frameworks* estão se tornando cada vez mais comuns e importantes. Eles são a maneira pela qual os sistemas orientados a objetos conseguem a maior reutilização. Aplicações orientadas a objetos maiores terminarão por consistir-se de camadas de *frameworks* que cooperam uns com os outros. A maior parte do projeto e do código da aplicação virá dos, ou será influenciada pelos *frameworks* que utiliza.

Normalmente o que diferencia um *framework* de uma biblioteca de classes é a chamada inversão de controle (*Inversion of Control* ou IoC). A inversão de controle permite que o *framework* chame funções que estão dento da aplicação. A IoC segue o princípio de *Hollywood*: "*Don't call us. We call you.*" (Não nos chame. Nós chamamos você).

A inversão de controle ocorre quando alguma classe da aplicação implementa uma interface ou estende uma classe abstrata do *framework*. Essa classe da aplicação estende alguma classe abstrata do *framework* que contém um algoritmo parcialmente implementado (classes *Template*). Esse algoritmo indica ao *framework* em que momentos ele deve assumir o controle e, em quais momentos a aplicação pode, ou deve, sobrescrever ou implementar os métodos desse algoritmo (métodos *Hooks*).

2.7.1. Tipos de *frameworks*

Os *frameworks* podem ser classificados a partir de diversos critérios. Um desses critérios é a forma como os *frameworks* são estendidos para gerar novas aplicações. A partir desse critério é possível dividir os *frameworks* em três grupos:

- *Frameworks* caixa-branca (*White-Box*);
- *Frameworks* caixa-preta (*Black-Box*);
- *Frameworks* caixa-cinza (*Gray-Box*).

Frameworks caixa-branca são instanciados usando herança. Costumam ser mais difíceis de usar, pois exigem que o desenvolvedor de aplicação conheça muito bem a estrutura interna do *framework*.

Frameworks caixa-preta são instanciados utilizando a composição de objetos. Normalmente são mais fáceis de usar do que os *frameworks* caixa-branca. Porém, são menos flexíveis e mais difíceis de desenvolver. A dificuldade de desenvolvimento ocorre devido à necessidade de identificação de muitos casos de uso. Isso é necessário para que o *framework* atenda ao domínio para o qual esta sendo construído.

Frameworks caixa-cinza possuem as características dos *frameworks* caixa-branca e caixa preta, no entanto, não apresentam as desvantagens que esses dois tipos possuem. De acordo com Silva (2006, p. 59),

> Esse tipo de *framework* permite a extensão tanto em termos de herança quanto em termos de composição, dependendo da necessidade da aplicação. Dessa forma os *frameworks* caixa-cinza possuem a flexibilidade e facilidade de extensão sem ter que prever muitos casos de uso e sem expor informações desnecessárias ao desenvolvedor.

Fayad *et al.* (1999 *apud* BARRETO) e Szyperski (1997 *apud* BARRETO) , além de classificar os *frameworks* de acordo com a forma como eles são estendidos, propõem a classificação a partir do escopo do *framework*:

- *Framework* de aplicação orientado a objeto;
- *Frameworks* de componentes.

Frameworks de aplicação orientados a objetos, ou apenas *frameworks* de aplicação, "geram famílias de aplicações orientadas a objetos. Seus pontos de extensão são definidos como classes abstratas ou interfaces, que são estendidas ou implementadas por cada instância da família de aplicações" (BARRETO, 2006, p.34). Disponibilizam funcionalidades básicas necessárias em uma aplicação, tais como interface gráfica, manipulação de dados, etc. Esse tipo de *framework* pode ser subdividido em:

- ***Frameworks* de infra-estrutura de sistemas**: facilitam o desenvolvimento de sistemas portáveis e eficientes, como sistemas operacionais.
- ***Frameworks* de integração de *middleware***: "são usados para integrar aplicações e componentes distribuídos. Esses *frameworks* escondem o baixo nível da comunicação entre componentes distribuídos" (BARRETO, 2006, p.35). Isso possibilita ao desenvolvedor trabalhar em ambientes distribuídos como se estivesse trabalhando em um ambiente não distribuído.
- ***Frameworks* de aplicações corporativas**: são desenvolvidos para um domínio de aplicação específico, como a área financeira ou de telecomunicações. Para o desenvolvimento deste tipo de *framework* é necessário a participação de especialistas

de domínio. Isso acaba tornando esse tipo de *framework* mais caro do que os de infra-estrutura de sistemas e integração de *middleware*. Esses *frameworks* também são conhecidos como *frameworks* de subsistemas ou *framework* de domínio. Eles encapsulam soluções para um domínio especifico e representam a parte vertical do domínio.

Segundo Szyperki (1997 *apud* BARRETO, 2006, p.36), um *framework* de componentes:

> É uma entidade de software que provê suporte a componentes que seguem um determinado modelo e possibilita que instâncias destes componentes sejam plugadas no *framework* de componentes. Ele estabelece as condições necessárias para um componente ser executado e regula a interação entre as instâncias destes componentes. Um *framework* de componentes pode ser único na aplicação, criando uma ilha de componentes ao seu redor, ou pode cooperar com outros componentes ou *frameworks* de componentes.

Frameworks de componentes diferem dos *frameworks* de aplicação no seguinte ponto: um *framework* de aplicação representa uma aplicação inacabada que dará origem a uma família de aplicações. Já os *frameworks* de componentes estabelecem um contrato para conectar componentes. Barreto (2006, p.36) utiliza uma analogia para explicar melhor essa diferença:

> Um *framework* de aplicação equivale ao quadro de uma bicicleta. Há lugares específicos (os pontos de extensão) onde o guidão, as rodas o banco e os pedais (implementações dos pontos de extensão) podem ser encaixados. Bicicletas diferentes (instâncias dos *frameworks*) podem ser compostas variando esses itens: bicicletas com marchas, com amortecedores, etc, mas apenas os pontos previstos podem variar e o produto final será sempre uma bicicleta. Por outro lado, *frameworks* de componentes podem ser vistos como uma placa de circuito integrado com *slots* onde os *chips* (componentes) podem ser encaixados para criar uma instância. Como os *frameworks* de componentes, as placas são utilizadas para compor soluções para diversos domínios como, por exemplo, uma placa de vídeo ou um modem.

2.7.2. Desenvolvimento, uso e evolução dos *frameworks*

O desenvolvimento de *frameworks* abrange um conjunto de etapas, não seqüenciais, que são repetidas até que a estrutura de classes obtida atenda aos requisitos de generalidade, alterabilidade e extensilidade. De acordo com Silva (2000), essas etapas são:

- **Generalização**: são identificadas estruturas comuns às aplicações analisadas de modo a obter uma estrutura de classes que generalize o domínio tratado.

- **Flexibilização**: são identificados os pontos da estrutura que devem ser mantidos flexíveis. Ou seja, identificar os pontos que poderão ser alterados para dar origem a novas aplicações.
- **Aplicação de metapadrões**[8]: após a identificação dos pontos flexíveis do *framework*, deve-se comparar os requisitos de flexibilização de cada ponto com padrões de projetos conhecidos. Se um padrão atender ao requisito de flexibilização, este deve ser incorporado ao *framework*. O uso de metrapadrões consiste em "transformar um procedimento geral em método *template*[9], cujo comportamento é flexibilizado através da dependência de métodos *hook*, que podem ter diferentes implementações" (SCHWEBEL, 2005, p. 60).
- **Aplicação de padrões de projeto**: são adicionadas as classes de um determinado padrão à estrutura de classes do *framework*. Ou, as classes existem são adaptadas para assumir as responsabilidades de um determinado padrão.
- **Aplicação de princípios práticos de orientação a objetos**: para tornar a estrutura de classes flexível são utilizados princípios orientados a objetos, como utilização de herança para reutilização de interface, reutilização de código através da composição de objetos, polimorfismo, etc.

Os *frameworks* são utilizados através da customização e extensão da estrutura oferecida. Os pontos variáveis de um *framework* são conhecidos como *hot spot*. As estruturas reutilizáveis do *framework*, ou seja, a arquitetura geral do *framework*, que inclui componentes básicos e seus relacionamentos, são conhecidos como pontos fixos (*frozen-spots*). Os *frozen-spots* não são customizados pelas aplicações construídas a partir do *framework*.

Quando maior a quantidade que pontos variáveis de um *framework*, menor será a reutilização. Pois, segundo Pree (1997 *apud* Silva, 2006, p.40),

> O potencial de reuso de um *framework* varia de acordo com o número de pontos fixos e variáveis, sendo que o potencial de reuso é menor se o número de pontos variáveis for grande em relação ao número de pontos fixos [...]. Isso significa que o *framework* oferece pouco a ser reusado.

Os *frameworks* podem ser utilizados em conjunto com outras abordagens para reutilização, tais como padrões de projeto, componentes de software, bibliotecas de classes, ou com outros *frameworks*, etc.

---

[8] Um metapadrão (*metappatern*) é uma abordagem complementar aos padrões de projeto. Os metapadrões representam aspectos em comum dos padrões de projeto. São utilizados para documentar o projeto de *frameworks* que possuem uma estrutura estável.

[9] Um método *template* disponibiliza um comportamento abstrato ou um fluxo de controle genérico através de código padrão, necessário para o funcionamento do *framework*, e chamadas a métodos *hook*.

O processo de desenvolvimento de software convencional envolve basicamente dois indivíduos: o desenvolvedor de aplicação e o usuário de aplicação. Os desenvolvedores são os responsáveis pelo levantamento de requisitos e construção da aplicação. Os usuários são as pessoas que interagem com a aplicação. Já o desenvolvimento e uso de *frameworks* envolve um individuo a mais, o desenvolvedor de *frameworks*. O papel do usuário de aplicação continua o mesmo, já o desenvolvedor de aplicação passa a ser o usuário do *framework*, ele executa as mesmas atividades descritas anteriormente, mas agora usando o *framework* para construir aplicações. O desenvolvedor, ou projetista, de *framework* é o responsável pela construção e documentação do *framework*.

Inicialmente, os *frameworks* costumam ser caixa-branca. Conforme os *frameworks* são utilizados, novos requisitos vão surgindo e modificações na estrutura do *framework* são necessárias para comportar esses requisitos. Conforme essas modificações vão ocorrendo, o *framework* tende a se transformar em caixa-preta e, depois de um tempo razoável de utilização e evolução, o *framework* acaba se tornando caixa-cinza.

As mudanças em um *framework* não são fáceis de serem realizadas. Segundo Schwebel (2005, p. 36),

> Se estas mudanças ocorrem enquanto a aplicação está sendo desenvolvida, não há grandes problemas, pois poderão ser tratadas normalmente durante o desenvolvimento da aplicação. Porém, mudanças como alteração de interfaces, novas classes e novas funcionalidades gerarão problemas que deverão ser tratados pelas aplicações já desenvolvidas com o *framework*.

Um ponto muito importante no processo de desenvolvimento, uso e evolução dos *frameworks* é a documentação. Sem documentação, utilizar e manter um *framework* pode se tornar uma tarefa muito difícil. Porém, não existem mecanismos padrões para documentação de *frameworks*.

Santos (2007) propõem a utilização de *cookbooks* e *hooks* para documentar *frameworks*. *Cookbooks* são tutoriais baseados em linguagem natural que ensinam como instanciar uma aplicação. Normalmente os *cookbooks* apresentam exemplos de como utilizar os componentes de uma aplicação. Além disso, as seções dos *cookbooks*, chamadas de receitas, descrevem como resolver problemas.

Hooks são extensões dos *cookbooks*. De acordo com Santos (2007, p. 30),

> Cada *hook* documenta um problema e um requisito [...] para a construção do *framework*, antecipando o desenvolvimento da aplicação e assim provendo um guia para se obter todo o potencial do *hook*. [...] Diferente dos *cookbooks*, os *hooks* trazem um formato que ajuda a descrever e organizar as informações de modo que possam ser facilmente utilizadas por uma ferramenta, além disso, os *hooks* expõem

> apenas os detalhes necessários para solucionar um problema no *framework*, o que o torna mais simples e fácil de entender.

2.7.3. Metodologias de desenvolvimento

O desenvolvimento de *frameworks* é diferente do desenvolvimento de sistemas de software convencionais. Isso ocorre porque um *framework* deve abranger todos os conceitos de um determinado domínio, enquanto um sistema convencional envolve apenas os conceitos especificados em seus requisitos. As metodologias de desenvolvimento tradicionais não suportam adequadamente o desenvolvimento de *frameworks*.

Hoje não existe uma metodologia padrão para o desenvolvimento de *frameworks*, no entanto, existem três propostas que são comumente utilizadas:

- Projeto dirigido por *Hot Spot* (*Hot Spot Driven Design*);
- Projeto dirigido *por exemplo* (*Example-Driven Design*);
- Metodologia de projeto da empresa *Taligent*[10].

Essas metodologias definem um processo genérico de desenvolvimento de *framework* e não detalham técnicas de modelagem ou o processo. Essas metodologias possuem alguns pontos em comum: busca de informações sobre o domínio tratado, a inexistência de técnicas de modelagem, utilização de *design patterns*, etc.

Segundo Schwebel (2005, p. 63),

> As metodologias de desenvolvimento de *frameworks* existentes estão, em sua maioria, voltadas a produzir código e não utilizam notação de alto nível, fazendo com que o resultado do processo de desenvolvimento seja o próprio código do *framework* – um modelo de domínio expresso através de uma linguagem de programação.

2.7.3.1. Projeto dirigido por *Hot Spot*

Frameworks devem possuir partes indefinidas que lhe permitem ser flexíveis e se adaptar a aplicações diferentes. *Hot spots* representam as partes flexíveis dos *frameworks*. O objetivo principal do projeto dirigido por *Hot Spot* é identificar essas partes flexíveis e, a partir daí, construir o *framework.*

A Figura 12 ilustra as etapas que fazem parte do projeto dirigido por *Hot Spot*.

---

[10] *Taligent* (*Talent and Intelligent*). Empresa fundada em 1992, a partir de uma parceria entre IBM e *Apple*. O objetivo da empresa era desenvolver uma nova geração de aplicações tendo como base a tecnologia orientada a objetos.

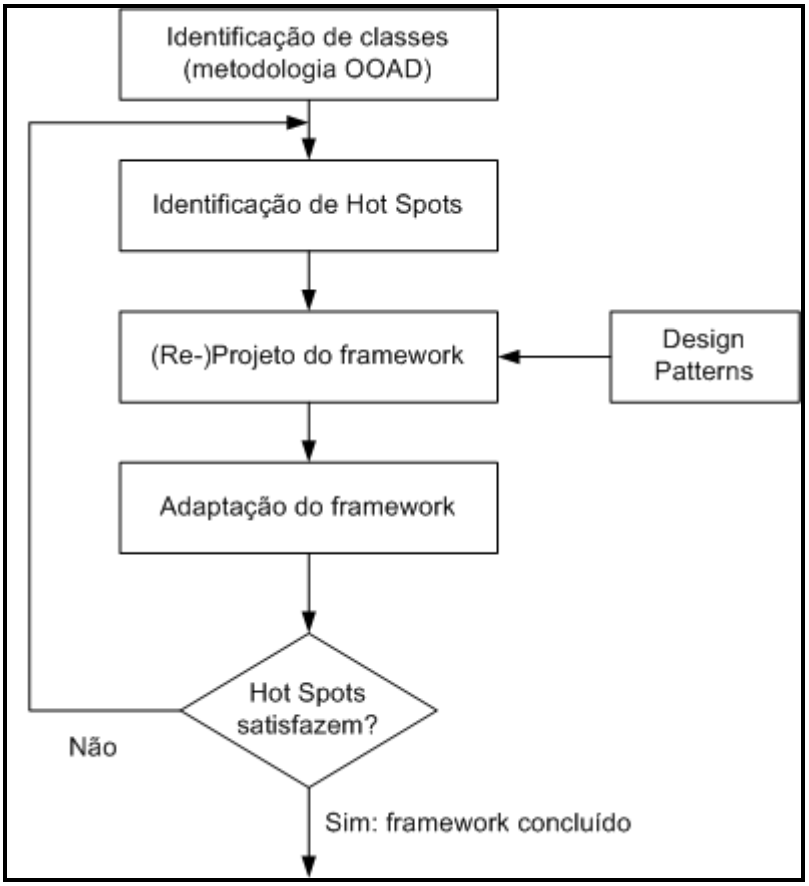

Figura 12 – Etapas do projeto dirigido por *hot spots*
Fonte: Silva (2000)

Segundo Silva (2000), essa metodologia de desenvolvimento utiliza as seguintes etapas:

1. **Identificação de classes**: semelhante ao que ocorre na metodologia orientada a objetos, com base nas informações sobre o domínio para o qual o *framework* será desenvolvido, o desenvolvedor define a estrutura classes.
2. **Identificação de *Hot Spots***: são identificados os aspectos que diferem de aplicação para aplicação e o grau de flexibilidade que cada caso deve oferecer. Os *hot spots* podem ser documentados em cartões *hot spot* semelhantes ao ilustrado na Figura 13.

Nome do *hot spot*	☐ Flexibilidade em tempo de execução
Descrição geral da semântica	
Comportamento do *hot spot* em pelos menos duas situações diferentes	

Figura 13 – Exemplo de cartão *hot spot*
Fonte: Silva (2000)

3. **(Re-)Projeto do *framework***: a estrutura de classes é alterada para suportar os *hot spots* e a flexibilidade requerida por cada *hot spot*. Essas alterações podem ocorrer através da aplicação de padrões de projeto. Essa etapa define como o *framework* será usado para construir aplicações.
4. **Adaptação do *framework***: com o auxilio de especialistas do domínio, a estrutura do *framework* é refinada. Se, após esse refinamento, o *framework* for considerado satisfatório, uma versão do *framework* está concluída. Caso contrário, retorna-se a etapa de identificação dos *hot spots*.

2.7.3.2. Projeto dirigido por exemplo

O desenvolvimento de *frameworks* para um domínio específico requer o aprendizado sobre esse domínio. No projeto dirigido por exemplo, esse aprendizado se dá através do estudo de aplicações conhecidas ou do desenvolvimento de aplicações para o domínio tratado. Segundo Silva (2000, p. 50),

> Porque as pessoas pensam de forma concreta, e não de forma abstrata, a abstração do domínio, que é o próprio *framework*, é obtida a partir da generalização de casos concretos – as aplicações. Abstrações são obtidas [...] a partir do exame de exemplos concretos: aspectos semelhantes de diferentes aplicações podem dar origem a classes abstratas que agrupam as semelhanças, cabendo às classes concretas do nível hierárquico inferior a especialização para satisfazer cada caso.

Segundo Silva (2002), as fases do projeto dirigido por exemplo são:

1. Análise de domínio
   1.1. Assimilação das abstrações conhecidas;
   1.2. Obtenção de exemplos de aplicações que podem ser construídas a partir do *framework* (recomenda-se no mínimo quatro);
   1.3. Avaliação de cada exemplo;
2. Projeto da hierarquia de classes do *framework*;
3. Teste do *framework* através da construção de exemplos de aplicações a partir do *framework*.

Por limitações econômicas ou de tempo, esses passos não costumam ser seguidos. Silva (2000, p.51), afirma que

> Em função da dificuldade prática de avaliar um mínimo de quatro aplicações já desenvolvidas para então começar a desenvolver o *framework*, é estabelecido um procedimento tido como adequado (e economicamente viável) para o

> desenvolvimento de um *framework*: a partir de duas aplicações similares que se necessite desenvolver, proceder paralelamente o desenvolvimento do *framework* e das duas aplicações, procurando maximizar a troca de informações entre os três desenvolvimentos.

2.7.3.3. Metodologia de projeto da empresa *Taligent*

A proposta da empresa *Taligent* consiste na definição de vários pequenos *frameworks* que atendam a partes especificas de um domínio. Esses pequenos *frameworks* substituiriam o desenvolvimento de um grande e complexo *framework* que comporte a abstração de todo o domínio tratado.

Nessa metodologia, de acordo com Silva (2000, p. 52),

> A visão de desenvolver um *framework* que cubra as características e necessidades de um domínio é substituída pela visão de produzir um conjunto de *frameworks* estruturalmente menores e mais simples, que usados conjuntamente, darão origem às aplicações. A justificativa para isso é que "pequenos *frameworks* são mais flexíveis e podem ser reutilizados mais freqüentemente". Assim, a ênfase passa a ser o desenvolvimento de *frameworks* pequenos e direcionados a aspectos específicos do domínio.

Essa metodologia também propõe tornar a utilização do *framework* o mais simples possível. Para isso procura minimizar a quantidade de código que o usuário do *framework* deve produzir através da:

- Disponibilização de implementações concretas que possam ser usadas diretamente;
- Minimização do número de classes que precisam ser criadas;
- Minimização do número de métodos que precisam ser sobrepostos.

Nesta metodologia, o primeiro passo para desenvolver um *framework* consiste em analisar o domínio do problema e identificar quais são os *frameworks* necessários. Para cada *framework* identificado é necessário: identificar as primeiras abstrações, projetar como será a interação entre os usuários e os *framework*, implementar, testar e refinar o projeto.

As etapas dessa metodologia são:

1. Identificação e caracterização do domínio do problema:
   1.1. Análise do domínio;
   1.2. Identificação dos *frameworks* necessários;
   1.3. Avaliação das soluções existentes;
   1.4. Identificação das abstrações principais;
   1.5. Identificação das responsabilidades dos *frameworks*;
   1.6. Validação das informações com o especialista de domínio;

2. Definição da arquitetura e do projeto:
   2.1. Refinamento da estrutura de classes;
   2.2. Aperfeiçoamento do projeto através da utilização de padrões de projeto;
   2.3. Validação das informações com o especialista do domínio;
3. Implementação do *framework*:
   3.1. Implementação das classes principais;
   3.2. Teste do *framework* através da geração de aplicações;
   3.3. Iteração para refinamento do projeto;
4. Desdobramento do *framework*:
   4.1. Gerar documentação necessária;
   4.2. Manter e atualizar o *framework.*

2.7.4. Benefícios decorrentes do desenvolvimento e utilização de *frameworks*

A utilização de *frameworks* traz uma série de benefícios para o desenvolvimento de software. Um *framework* torna mais fácil desenvolver aplicações, aumenta a produtividade dos desenvolvedores e a qualidade do software. *Frameworks* apresentam resultados em longo prazo e capturam o conhecimento de especialistas de domínio.

Além disso, Fayad e Schmidt (1997), afirmam que existem quatro benefícios principais decorrentes do desenvolvimento e uso de *frameworks*:

- **Modularidade**: *frameworks* encapsulam detalhes de implementação em interfaces estáveis. Isso melhora a qualidade do software através da identificação do impacto de mudanças na implementação. Os pontos de mudanças no *framework* são localizados, diminuindo o tempo necessário para entender e manter aplicações e *frameworks*.
- **Reusabilidade**: os *frameworks* definem componentes genéricos que podem ser utilizados para construir diversas aplicações. Além disso, também capturam o conhecimento dos desenvolvedores em determinado domínio.
- **Extensibilidade**: um *framework* aumenta a capacidade de extensão através da utilização de métodos *hook* para estender suas interfaces. Os métodos *hook* separam as interfaces e comportamentos de um domínio de aplicação da variação necessária por uma aplicação em um determinado contexto.

- **Inversão de controle**: permite ao *framework* assumir o fluxo de controle da aplicação em tempo de execução. Dessa forma, o desenvolvedor se ocupa apenas em implementar as funcionalidades necessárias.

2.7.5. Desafios decorrentes do desenvolvimento e utilização de *frameworks*

O desenvolvimento e uso de *frameworks* apresenta uma série de benefícios. No entanto, também existem alguns desafios que os desenvolvedores e usuários de *framework* vão enfrentar.

Frameworks são mais difíceis de desenvolver do que aplicações convencionais. Também são difíceis de aprender a utilizar. Manter um *framework* pode se tornar um trabalho árduo se ele não for bem estruturado e documentado desde o início.

Fayad e Schmidt (1997) citam alguns desafios decorrentes do desenvolvimento e utilização de *frameworks*:

- **Esforço de desenvolvimento**: desenvolver *frameworks* de alta qualidade, extensíveis e reutilizáveis é difícil. O conhecimento necessário para o desenvolvimento de *frameworks* bem sucedidos, normalmente está restrito a desenvolvedores experientes e especialistas de domínio.
- **Curva de aprendizado**: aprender a utilizar *frameworks* é uma tarefa difícil. Nem sempre existe documentação suficiente para auxiliar este aprendizado. E, no caso de *frameworks* grandes e complexos, muitas vezes, são necessários cursos ou treinamentos para aprender a utilizá-los. Os custos decorrentes do aprendizado de *frameworks* só é justificável se o *framework* for utilizado em diversos projetos.
- **Integração**: o desenvolvimento de aplicação pode ser baseado na integração de diversos *frameworks*, bibliotecas de classes, sistemas legados e componentes existentes. Se os *frameworks* não forem projetados tendo em mente essa integração, podem ocorrer sérios problemas durante o desenvolvimento de aplicações.
- **Manutenção**: os requisitos das aplicações costumam mudar com freqüência, conseqüentemente, os requisitos dos *frameworks* também mudam. A manutenção do *framework* envolve a inclusão, modificação ou remoção de funcionalidades. Para isso, é necessário um grande conhecimento dos componentes do *framework* e dos seus inter-relacionamentos.

- **Validação e remoção de defeitos**: apesar de *frameworks* bem projetados diminuírem o impacto de defeitos nas aplicações, a validação e depuração de defeitos nessas aplicações pode ser mais difícil do que em aplicações que não utilizam *frameworks*. Isso corre devido à oscilação do fluxo de controle entre a aplicação e o *framework*.
- **Eficiência**: aplicações que utilizam *frameworks* podem ter desempenho inferior. Isso ocorre devido aos níveis de indireção necessários para manter o *framework* genérico. Normalmente aplicações genéricas apresentam desempenho inferior ao de aplicações específicas.
- **Falta de padronização**: não existem padrões para o desenvolvimento, uso, manutenção e documentação de *frameworks*.

2.8. Web 2.0 e RIA (*Rich Internet Application*)

Dedes que o *World Wilde Web* (www, ou simplesmente *web*) surgiu, no inicio da década de 1990, ele vem passando por uma constante evolução. O surgimento da *web* modificou completamente a forma como as pessoas se comunicação e a maneira como as organizações se comportam no mercado.

A empresa *O'Reilly Media* foi a primeira a utilizar o termo *Web* 2.0 para designar essa nova fase da *web*. Isso ocorreu em 2004 em uma conferência entre a *O'Reilly Media* e a *MediaLive International*. A *Web* 2.0 representa uma nova geração da *web* onde páginas estáticas foram substituídas por conteúdo dinâmico. O ponto central da *Web* 2.0 é a troca de informações entre usuários e os serviços disponíveis na *web*. Essa nova forma de interação entre usuários e a *web* é conhecida como *Web* 2.0.

De acordo com Platt (2007, p.2),

> Recentemente, os modelos antigos que estabelecem como as pessoas publicam e consomem as informações na *Web* transformaram-se de modo radical. Em lugar de simplesmente visualizar as informações em páginas da *Web* estáticas, os usuários agora publicam conteúdo próprio nos *blogs*, em *wikis* e *sites* que compartilham fotos e vídeos. As pessoas estabelecem colaboração, listas de discussões e comunidades online; combinam dados, conteúdo e serviços de várias fontes para criar experiências e aplicativos personalizados.

O'Reilly (2005, p.1) define a *Web* 2.0 como "um conjunto de princípios e práticas que interligam um verdadeiro sistema solar de sites que demonstram alguns ou todos esses princípios e que estão a distâncias variadas do centro."

Rich Internet Application (RIA) é um conceito inovador para desenvolvimento de aplicações *web*. Sistemas que são desenvolvidos utilizando este conceito unem as funcionalidades disponíveis em aplicativos *desktops* com as facilidades oferecidas pelo ambiente *web*. Algumas ferramentas utilizadas para o desenvolvimento de RIAs são *Adobe Flex, Microsoft Silverligh, OpenLazzlo, JavaScript, JavaFX, Google Web Toolkit,* etc.

2.9. Processo Unificado

O Processo Unificado (*Unified Process* – UP) é dirigido por casos de uso, centrado na arquitetura e iterativo e incremental. Essa metodologia de desenvolvimento é definida por Scott (2003, p.19) como "um conjunto de atividades executadas para transformar um conjunto de requisitos do cliente em um sistema de software". Porém, Scott (2003) também afirma que o "Processo Unificado é uma estrutura genérica de processo que pode ser customizado adicionando-se ou removendo-se atividades com base nas necessidades e nos recursos disponíveis para um projeto".

Um exemplo de especialização do Processo Unificado é o RUP (*Rational Unified Process*). Para Scott (2003, p. 134), o RUP "é uma instância específica e (altamente) detalhada do Processo Unificado".

O Processo Unificado é dirigido por casos de uso porque utiliza os casos de uso para direcionar todas as atividades de desenvolvimento, incluindo desde a identificação e negociação dos requisitos até a aceitação do código fonte.

No Processo Unificado a arquitetura é considerada base fundamental sobre a qual o sistema será construído. Por isso, deve ser um dos principais pontos de preocupação durante o desenvolvimento. Juntamente com os casos de uso, a arquitetura deve orientar todos os aspectos do sistema.

O Processo Unificado é iterativo e incremental. Cada iteração resulta em uma nova versão do sistema. Essas versões devem oferecer "uma melhora incremental sobre a anterior, motivo pelo qual a iteração é chamada de incremento" (SCOTT, 2003, p. 24).

De acordo com Waslawick (2004), o Processo Unificado comporta as etapas de estudo de viabilidade, análise de requisitos, análise de domínio, e projeto em múltiplas camadas. No entanto, essas etapas aparecem no Processo Unificado organizadas em quatro fases: concepção, elaboração, construção e transição.

Durante a fase de concepção são levantados os primeiros requisitos funcionais do sistema. O objetivo principal é realizar a análise econômica para o sistema, estabelecendo a viabilidade do sistema proposto. Segundo Scott (2003, p. 84) a versão inicial do sistema "é a justificativa para comprometer-se com o projeto de desenvolvimento, cuja continuidade as versões posteriores pretendem justificar". Alguns dos passos dessa fase são:

- Estabelecer o escopo do sistema;
- Explorar os requisitos funcionais e não-funcionais de alto nível;
- Estruturar uma arquitetura candidata;
- Identificar os riscos mais críticos que o projeto enfrentará;
- Especificar valores, como o retorno sobre o investimento ou os ganhos em produtividade.

Na fase de elaboração são identificados os requisitos funcionais não determinados na fase de concepção e, estabelecida uma base arquitetônica para o desenvolvimento do sistema. Também se expande e refina-se a análise econômica e abordam-se questões sobre orçamento e cronograma. Esta fase estabelece a capacidade para a construção do sistema. De acordo com Scott (2003), os passos dessa fase são:

- Desenvolver os casos de uso relacionados aos requisitos funcionais que não foram tratados durante a concepção;
- Criar uma arquitetura que abranja os elementos mais significativos do sistema;
- Focar-se nos riscos significativos que podem causar atrasos e gastos maiores que o previsto;
- Definir os níveis de qualidade do sistema;
- Detalhar a análise econômica e o plano de projeto.

A fase de construção tem como objetivo principal construir uma versão operacional do sistema que possa ser utilizada em *ambiente beta*[11] de clientes. Os passos principais desta fase são:

- Concluir os modelos de casos de uso, análise, projeto, instalação, implementação e teste;
- Alterar a arquitetura conforme necessário;
- Monitorar os riscos críticos e os riscos significativos.

[11] Ambientes de teste. Os usuários do sistema irão utilizá-lo e reportar erros, ou melhorias, à equipe de desenvolvimento, que, por sua vez, corrigirá as falhas e estudará a viabilidade de implementação das melhorias.

Durante a fase de transição os clientes betas recebem uma versão completamente funcional do sistema, enquanto se continua a corrigir as falhas reportadas pelos clientes. Os passos dessa fase são:

- Preparar a divulgação do sistema;
- Preparar a documentação do sistema;
- Divulgar o sistema;
- Adaptar o sistema para ser executado no ambiente do cliente;
- Obter informações sobre falhas do sistema e corrigir tais falhas.

Dentro do Processo Unificado, "cinco *workflows* atravessam o conjunto das quatro fases: requisitos, análise, projeto, implementação e teste. Cada workflow é um conjunto de atividades que vários membros do projeto executam." (SCOTT, 2003, p.29).

Segundo Scott (2003, p. 34), "o princípio fundamental do *workflow* de requisitos é trabalhar para construir o sistema certo". Este *workflow* visa à construção do modelo de casos de uso que representará os requisitos funcionais do sistema. Esse modelo ajuda a compreender quais são as capacidades do sistema e as condições que o sistema deve satisfazer. Esse *workflow* também serve como alicerce para qualquer trabalho de desenvolvimento.

As atividades do *workflow* de análise visam à construção do modelo de análise, este ajuda os desenvolvedores a refinar e estruturar os requisitos funcionais capturados no modelo de casos de uso. O principal objetivo é entender os requisitos dos clientes.

O *workflow* de projeto visa à construção do modelo de projeto. Este modelo descreve quais as realizações físicas dos casos de uso a partir do modelo de casos de uso e do modelo de análise. Este *workflow* também serve como uma abstração do modelo de implementação e foca-se no modelo de instalação, que define a organização física do sistema.

As atividades do *workflow* de implementação visam à construção do modelo de implementação, que descreve como os elementos do modelo de projeto são empacotados em componentes de software. O objetivo principal desse *workflow* é construir uma versão operacional do sistema que possa ser entregue aos clientes.

O *workflow* de teste visa à construção do modelo de testes, que descreve como os testes de integração e de sistema serão realizados. Esse workflow também contém os resultados de todos os níveis de teste. O objetivo principal desse *workflow* é garantir a qualidade do sistema que será entregue.

2.10. UML (*Unified Modeling Language*)

A modelagem é uma das principais atividades do desenvolvimento de um bom software. A modelagem permite especificar o que deve ser feito e, em alguns casos, como deve ser feito. A construção de modelos ajuda a entender a estrutura e o comportamento dos componentes do sistema, permite visualizar aspectos arquiteturais e ter uma compreensão geral do sistema pretendido.

As linguagens de programação não fornecem um nível de abstração alto o suficiente para discutir determinados aspectos de um software. Para minimizar este problema, surgiram diversas linguagens de modelagem gráfica. Essas linguagens procuram tratar os elementos que compõem um software em um nível de abstração que qualquer pessoa envolvida com o projeto consiga compreendê-lo.

A UML (*Unified Modeling Language*) surgiu da unificação de diversas linguagens de modelagem gráfica utilizadas nos anos oitenta e noventa. De acordo com Melo (2002, p. 30),

> Com o passar do tempo, cada método ganhava uma fatia diferente do mercado. Juntamente com a divisão do mercado, pairava uma acirrada competição. Tentativas de padronização foram propostas, mas não obtiveram sucesso. Por volta de 1993, os métodos que mais cresciam no mercado eram: Booch'93, OMT-2 e OOSE. Todavia, apesar das semelhanças, existiam pontos significativos e fortes em cada método. Resumidamente, o OOSE possuía seu foco em casos de uso (*use cases*), provendo excelente suporte á engenharia de negócios e análise de requisitos. OMT-2 era expressivo na fase de análise dos sistemas de informação. Booch'93 já se destacava na fase de projeto. Ao invés de seguir a linha dos primeiros autores (que procuram redefinir ou estender métodos existentes), Booch, Rumbaugh e Jacobson decidiram unir forças e criar um método único. Seus métodos já estavam evoluindo um em direção ao outro, de maneira independente. Seria mais sensato essa evolução prosseguir de forma conjunta do que individualmente.

Os esforços para unificar a UML se iniciaram em 1994, quando *James Rumbaugh* deixou a *General Eletric* para se unir a *Grady Booch* na *Rational Software*. O objetivo era unificar os métodos *Booch* e OMT. Em 1995, a *Rational* lançou a versão 0.8, do então chamado, Método Unificado. Nesta época *Jacobson* se juntou a equipe com o intuito de incorporar o método OOSE ao Método Unificado. Em 1996 o Método Unificado passou a se chamar UML e foram lançadas duas versões: UML 0.9 e UML 0.91. Ainda em 1996, a OMG fez um requerimento de proposta de padronização (*Request for Proposals* – RFP). Em resposta a esta proposta, em 1997, a *Rational* lançou a versão 1.0 da UML. Nesse mesmo ano a UML, na versão 1.1, foi aceita como padrão pelo OMG. Deste então o OMG é o responsável pela manutenção da UML.

A UML "é uma família de notações gráficas, apoiada por um metamodelo único, que ajuda na descrição e no projeto de sistemas de software, particularmente daqueles construídos utilizando o estilo orientado a objetos" (FOWLER, 2005, p. 25). Ou, nas palavras de Booch, Jacobson e Rumbaugh (2005, p. 14) a UML "é uma linguagem destinada a: visualizar, especificar, construir e documentar os artefatos de um sistema complexo de software."

Ao utilizar a UML para modelar um sistema, são construídos modelos que representam diferentes pontos de vista. Cada modelo descreve "um aspecto particular da mesma solução. Esses pontos de vista são denominados visões, isto é, perspectivas ou focos em ângulos diferentes de um mesmo sistema" (LIMA, 2005, p. 33).

Cada visão utiliza um conjunto de notações, elementos, que permitem compreender o sistema através da perspectiva do usuário, ou do analista, ou do projetista, ou do programador, etc. De acordo com Lima (2005), a UML permite representar um sistema em cinco visões:

- **Visão do caso de uso**: é a base para as demais visões. Permite visualizar o sistema através da perspectiva do usuário. Lima (2005, p. 34) afirma que essa visão "serve como um contrato entre o cliente e o desenvolvedor porque mostra conceitualmente o conjunto de funções que o sistema deve executar para atender aos requisitos".
- **Visão lógica**: define as classes, subsistemas e pacotes lógicos do sistema. É o ponto de vista arquitetônico. Representa o ponto de vista dos analistas, projetistas e desenvolvedores do sistema.
- **Visão do processo**: captura aspectos estáticos e dinâmicos do sistema através de diagramas de interação, atividades e estados. Representa o ponto de vista do analista e do desenvolvedor do sistema.
- **Visão de implementação**: especifica subsistemas e suas dependências e componentes, e especifica como eles serão organizados em camadas e hierarquias. Representa o ponto de vista dos analistas e desenvolvedores do sistema.
- **Visão de implantação**: representa "a distribuição física do sistema através do conjunto de nós[12] do ambiente em que ele vai ser executado [...] incluindo distribuição de processos e *threads*" (LIMA, 2005, p. 35).

As visões descritas à cima são abstrações de modelos. Cada uma delas se preocupa com um aspecto em particular. Dessa forma, as visões UML facilitam a modelagem de sistema e melhoram a manutenção dos modelos.

[12] Exemplos de nós são: servidores, terminais cliente, mainframes, etc.

2.10.1. Diagramas UML

Sistemas de grande porte não podem ser totalmente compreendidos através de uma única perspectiva. Por isso a UML disponibiliza uma série de diagramas que permite modelar um sistema a partir de diferentes pontos de vista.

Conforme ilustrado na Figura 14, a UML 2.0 disponibiliza um conjunto de diagramas distribuídos entre diagramas de estrutura e diagramas de comportamento. Os diagramas de estrutura fornecem uma visão de como os elementos que compõem o sistema são organizados. Já os diagramas de comportamento permitem visualizar como cada elemento deverá se comportar no sistema.

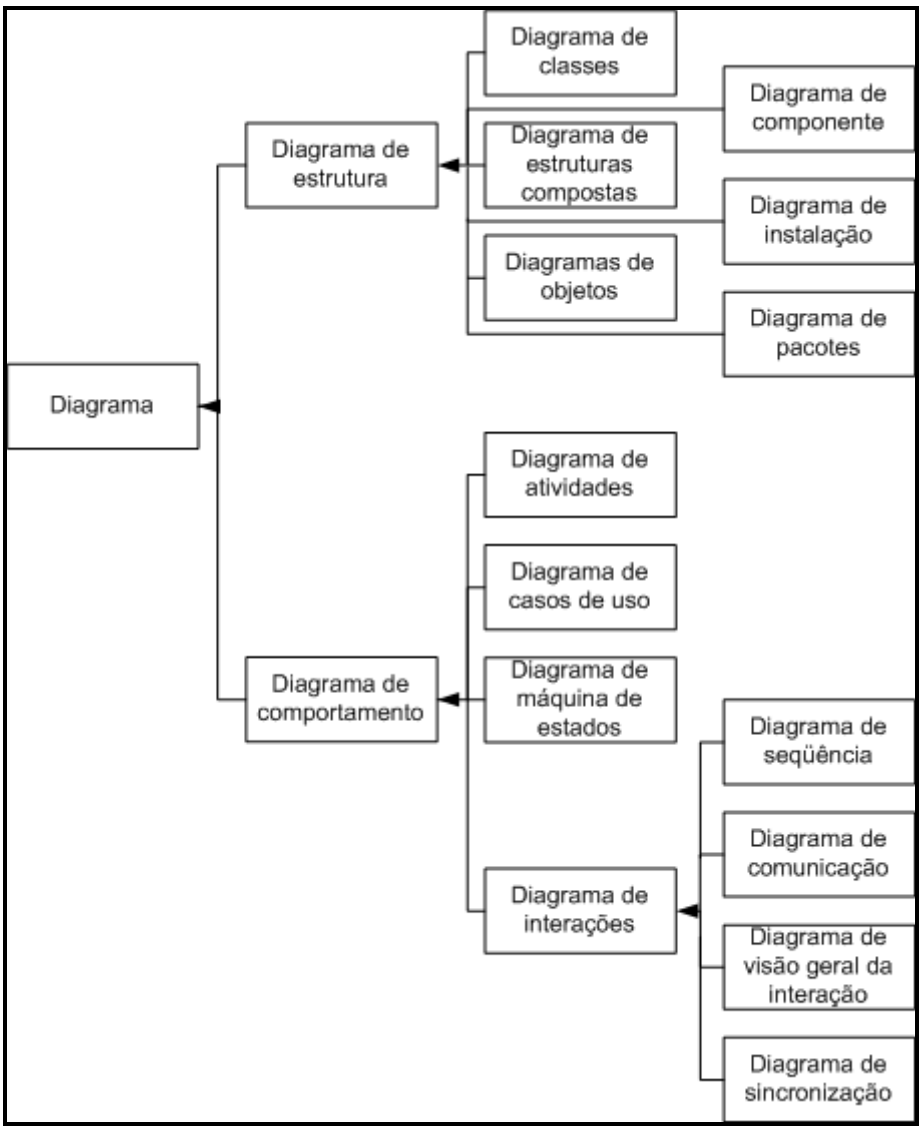

Figura 14 – Classificação dos tipos de diagrama da UML
Fonte: Fowler (2005)

A Tabela 11 apresenta uma breve descrição dos diagramas UML.

Tabela 11 – Tipos de diagramas UML

Diagrama	Objetivo	Linhagem
Atividades	Comportamento procedimental e paralelo	Na UML 1
Classes	Classes, características e relacionamentos	Na UML 1
Comunicação	Interação entre objetos; ênfase nas ligações	Diagrama de colaboração da UML 1
Componentes	Estrutura e conexão de componentes	Na UML 1
Estruturas compostas	Decomposição de uma classe em tempo de execução	Novidade da UML 2
Distribuição	Distribuição de artefatos nos nós	Na UML 1
Visão geral de integração	Mistura de diagrama de seqüência e de atividades	Novidade da UML 2
Objetos	Exemplo de configurações de instâncias	Extra-oficialmente na UML 1
Pacotes	Estrutura hierárquica em tempo de compilação	Extra-oficialmente na UML 1
Seqüência	Interação entre objetos; ênfase na seqüência	Na UML 1
Máquinas de estado	Como os eventos alteram um objeto no decorrer de sua vida	Na UML 1
Sincronismos	Interação entre objetos; ênfase no sincronismo	Novidade da UML 2
Casos de uso	Como os usuários interagem com o sistema	Na UML 1

Fonte: Fowler (2005)

Cada elemento que compõe os diagramas UML possui um propósito especifico. Além disso, os diagramas também possuem regras e notações que definem determinadas situações. A maior parte dos diagramas apresenta nós (símbolos ou construções gráficas) ligados entre si através de uma linha, representando um relacionamento. Porém um relacionamento também pode "ser expresso na forma de recipientes, em que um símbolo está contido em outro, ou de

anexo, em que símbolo aparece próximo a outro, geralmente ligado por uma linha pontilhada indicando uma nota explicativa ou comentário" (LIMA, 2005, p.36).

2.10.1.1. Diagrama de classes

Os diagramas de classe mostram um conjunto de classes, interfaces e colaborações. Além disso, mostram os atributos e métodos de cada classe e as restrições aplicadas à forma como os objetos são conectados. Ou, nas palavras de Blaha e Rumbaugh (2006, p. 25),

> Diagramas de classes oferecem uma notação gráfica para modelar classes e seus relacionamentos, descrevendo assim possíveis objetos. Os diagramas de classes são úteis para a modelagem abstrata e para o projeto de programas reais. Eles são concisos, fáceis de entender e funcionam bem na prática.

Os elementos básicos de um diagrama de classes são:

- **Classes**: são representados por caixas retangulares divididas em três partes. A primeira contém o nome da classe, a segunda os atributos e a terceira as operações.
- **Atributos**: são apresentados dentro da segunda partição de uma caixa de classe no seguinte formato: *visibilidade nome : tipo multiplicidade = valor-default {lista de propriedades}*. Apenas o nome é obrigatório, as demais propriedades são opcionais.
- **Operações**: são apresentadas dentro da terceira partição de uma caixa de classe no seguinte formato: *visibilidade nome(lista de parâmetros) : tipo de retorno {lista de propriedades}*. Apenas o nome é obrigatório.
- **Relacionamento**: pode ocorrer de diversas maneiras, as mais comuns são:
  - Associação;
  - Agregação;
  - Composição;
  - Generalização;
  - Dependência.
- **Classes de associação**: representam atributos, operações e qualquer outra característica que pertença ao relacionamento existente entre classes.
- **Multiplicidade**: indica a quantidade de objetos que podem preencher uma determinada propriedade. Mais especificamente, "representam o número de instâncias de uma classe que podem se relacionar com uma única instância de uma classe associada" (BLAHA e ROUMBAUGH, 2006, p.31). Por exemplo, um Cliente pode fazer muitos Pedidos, mas um Pedido só pode conter um Cliente.

- **Notas e comentários**: em um diagrama de classes uma nota representa um comentário. As notas podem aparecer isoladas ou associadas a um elemento através de uma linha tracejada.

2.10.1.2. Diagrama de casos de uso

As interações entre sistemas e usuários podem ser modeladas em diferentes níveis de abstração. No nível mais alto essas interações são modeladas utilizando diagramas de casos de uso. Os casos de uso são utilizados para capturar os requisitos funcionais de um sistema. Eles representam as interações existentes entre os atores (usuários, sistemas externos, hardware, etc) e o sistema. Essas interações são representadas através de cenários. Cenários são seqüências de passos que descrevem as interações dos usuários com o sistema. De acordo com Booch, Jacobson e Rumbaugh (2000, p. 231),

> Os diagramas de casos de uso são importantes para visualizar, especificar e documentar o comportamento de um elemento. Esses diagramas fazem com que sistemas, subsistemas e classes fiquem acessíveis e compreensíveis, por apresentarem uma visão externa sobre como esses elementos podem ser utilizados no contexto. Os diagramas de casos de uso também são importantes para testar sistemas executáveis por meio de engenharia de produção e para compreendê-los por meio de engenharia reversa.

A parte mais importante de um caso de uso não é sua representação gráfica, e sim seu conteúdo. Porém, a UML não fornece nenhuma maneira padronizada de descrever o conteúdo dos casos de uso.

Os diagramas de casos de uso são compostos por:

- **Casos de uso**: são as descrições de como ocorrem às interações entre os usuários e o sistema;
- **Atores**: são entidades externas com interagem com o sistema, podem ser os usuários finais, algum tipo de hardware ou outros sistemas;
- **Relacionamentos**: demonstram os relacionamentos existentes entre casos de uso e atores.

2.10.1.3. Diagrama de seqüência

Os diagramas de seqüência são utilizados para descrever como objetos interagem para realizar algum comportamento. Cada digrama de casos de uso pode exigir mais de um

diagrama de seqüência para descrever seu comportamento. No entanto, cada diagrama de seqüência representa o comportamento de um caso de uso. Segundo Blaha e Rumbaugh (2006, p.136),

> Os diagramas de seqüência fornecem mais detalhes e mostram as mensagens trocadas entre um conjunto de objetos ao longo do tempo. As mensagens incluem sinais assíncronos e chamadas de procedimentos. Os diagramas de seqüência são bons para mostrar as seqüências de comportamentos vistas pelos usuários de um sistema.

Os diagramas de seqüência dão ênfase à ordenação temporal das mensagens trocadas entre os objetos. Graficamente são representados por uma tabela que mostra os objetos no eixo X e mensagens no eixo Y. Cada objeto possui uma linha de vida, que representa o tempo que o objeto existe durante a interação. Graficamente a linha de vida é representada por uma linha tracejada vertical. Os objetos também têm foco de controle, que determinam quando o objeto esta realizando alguma operação. O foco de controle é representado, graficamente, por um retângulo em cima da linha de vida do objeto.

2.10.1.4. Diagrama de atividades

Os diagramas de atividade são utilizados para modelar aspectos dinâmicos de um sistema. Esses diagramas mostram as seqüências de atividades necessárias para realizar um processo complexo. Ao contrário dos diagramas de seqüência, cujo foco principal são os objetos, os diagramas de atividades se concentram nas operações. Segundo Fowler (2005, p. 118),

> Os diagramas de atividades são uma técnica para descrever lógica de procedimento, processo de negócio e fluxo de trabalho. De várias formas, eles desempenham um papel semelhante aos fluxogramas, mas a principal diferença entre eles e a notação de fluxograma é que os diagramas suportam comportamento paralelo.

2.10.1.5. Diagrama de comunicação

Antes da UML 2.0, esses diagramas eram conhecidos como diagramas de colaboração. Os diagramas de comunicação são mais uma maneira de visualizar as interações do sistema. No entanto, esses diagramas estão focados nos vínculos existentes entre os participantes da interação. Segundo Fowler (2005, p.129),

> Os diagramas de comunicação, um tipo de diagrama de interação, enfatizam os vínculos de dados entre os vários participantes na interação. Em vez de desenhar cada participante como uma linha de vida e mostrar a seqüência de mensagens por meio da direção vertical, como fazem os diagramas de seqüência, o diagrama de comunicação permite livre posicionamento dos participantes, permite desenhar vínculos para mostrar como eles se conectam e usa numeração para mostrar a seqüência de mensagens.

2.10.1.6. Diagrama de objetos

Os diagramas de objetos permitem modelar instâncias de classes em um determinado ponto do tempo. Em estruturas muito complexas, visualizar apenas o estado de um único objeto pode não ajudar a compreender a estrutura do sistema. Esses diagramas são úteis em situações onde a estrutura de um diagrama de classes é difícil de compreender.

Diagramas de objetos bem-estruturados têm o foco voltado para um único aspecto da visão estática do sistema. Eles possuem apenas os elementos essenciais para a compreensão dessa visão.

2.10.1.7. Diagrama de máquinas de estado

Uma máquina de estados permite modelar o comportamento de um objeto. A máquina de estados determina a seqüência de estados que um objeto pode passar durante seu ciclo de vida.

Os diagramas de máquinas de estados são compostos por:

- **Estados**: representam um momento durante o qual o objeto realiza alguma operação ou atende a determinada condição;
- **Transições**: representam as associações entre os estados de um objeto;
- **Eventos**: representa algo que pode alterar o estado de um objeto.

Segundo Booch, Jacobson e Rumbaugh (2000, p. 285),

> As máquinas de estados são empregadas para modelagem dos aspectos dinâmicos de um sistema. Na maior parte, isso envolve a especificação do tempo de vida das instâncias de uma classe, um caso de uso ou um sistema inteiro. Essas instâncias poderão responder a eventos como sinais, operações ou passagem de tempo. Quando um evento ocorre, alguma atividade acontecerá, dependendo do estado atual do objeto. Uma atividade é uma execução não-atômica em andamento em uma máquina de estados. As atividades acabam resultando em alguma ação, formada por computações atômicas executáveis que resultam em uma alteração do estado do modelo ou no retorno de um valor. O estado de um objeto é a condição ou situação durante a vida de um objeto durante a qual ele satisfaz alguma condição, realiza alguma atividade ou aguarda algum evento.

Os diagramas de máquinas de estados devem ser simples e possuir um contexto claro. As transições e estados devem ser nomeadas de acordo com as convenções do sistema.

2.10.1.8. Diagrama de pacotes

Os pacotes permitem agrupar diversos elementos da UML. Porém, a utilização mais comum é para agrupar classes, embora também sejam utilizados para agrupar casos de uso. O objetivo da utilização de pacotes é organizar os elementos em grupos.

De acordo com Booch, Jacobson e Rumbaugh (2000, p. 170),

> Um pacote pode conter outros elementos, incluindo classes, interfaces, componentes, nós, colaborações, casos de uso, diagramas e até mesmo outros pacotes. A propriedade de elementos é um relacionamento composto, significando que os elementos são declarados no pacote. Se o pacote dor destruído, os elementos serão destruídos. Cada elemento pertence, de maneira única, a exatamente um pacote.

Quando os pacotes são bem organizados eles fornecem uma fronteira para um grupo de elementos inter-relacionados. Além disso, eles não podem ser nem muito grande, nem muito pequeno. Pacotes muito grandes devem ser divididos e pacotes muito pequenos devem ser combinados com outros pacotes.

2.10.1.9. Diagramas de componentes

Diagramas de componentes permitem visualizar a organização e dependência entre os componentes de um sistema. Eles modelam uma visão estática do sistema.

Para Booch, Jacobson e Rumbaugh (2000, p. 387), os diagramas de componentes não "são importantes somente para visualizar, especificar e documentar sistemas baseados em componentes, mas também para a elaboração de sistemas executáveis por meio de engenharia de produção e reversa".

2.10.2. Mecanismos de extensão

Para que a UML continue existindo, ela precisa ser capaz de evoluir e se adaptar a novos domínios. Para isso, a UML disponibiliza três mecanismos de extensão:

- **Estereótipos**: “permite criar um novo elemento diferente mudando a semântica de um elemento do modelo UML existente. Basicamente, isso leva ao acréscimo de um novo vocabulário à UML” (AHMED e UMRYSH, 2002, p. 29).
- **Valores marcados**: permite definir e associar propriedades a um elemento do modelo através de um par *tag* e valor no formato *tag=valor*.
- **Limites (ou restrições)**: permite especificar restrições e relações que não podem ser expressas. Ele é expresso através de uma *string* colocada entre chaves: {limite}

Um dos objetivos da UML sempre foi “fornecer um vocabulário comum, portanto estender a linguagem a gosto de qualquer pessoa é se opor à finalidade assim com ao espírito da UML” (AHMED e UMRYSH, 2002, p. 31). Dessa forma, quando surge a necessidade de adaptar a UML a uma necessidade específica, sugere-se a criação de um novo perfil UML. Esse perfil não estende a UML. Um perfil UML usa os mecanismos de extensão para determinar uma maneira comum de usar UML no contexto de um novo domínio. Ou seja, “um perfil UML é basicamente uma coleção de estereótipos, limites, valores marcados e ícones junto com convenções para usá-los no novo domínio” (AHMED e UMRYSH, 2002, p. 31).

Apesar de a UML disponibilizar uma série de diagramas, ela não fornece uma notação adequada para modelagem de *frameworks*. No entanto, existe uma extensão, ou perfil, UML que utiliza os mecanismos de extensão para disponibilizar uma série de construções úteis para modelar *frameworks*. Esse perfil é conhecido com UML-F.

2.11. Linguagens de programação

Uma linguagem de programação é uma ferramenta de desenvolvimento que permite ao programador escrever instruções que serão compiladas, ou interpretadas, gerando um software executável.

Devido à evolução das linguagens de programação, elas podem ser classificadas em:

- **Linguagens de primeira geração ou linguagens de máquina**: são as linguagens nativas entendidas por um sistema computacional. São linguagens muito complexas para os seres humanos e apresentam a desvantagem de serem dependentes da máquina. Ou seja, a linguagem de uma determinada máquina só pode ser utilizada para aquela máquina e nenhuma outra. Segundo Audy, Andrade e Cidral (2005, p. 173), “essa programação primitiva implicava a determinação das seqüências de bits

correspondentes às instruções e dados a serem processados". A linguagem de máquina é o mais baixo nível de abstração em que um software é representado.

- **Linguagens de segunda geração ou linguagens *assembly***: devido à complexidade das linguagens de máquina, os programadores começaram a utilizar abreviações em inglês para representar as instruções necessárias para realização das operações, dando origem as chamadas linguagens *assembly*. Essas instruções eram traduzidas para linguagem de máquina através de um programa montador (*assembler*). Alguns exemplos desta geração são: *Fortran, Cobol, Algol* e *Basic*.
- **Linguagens de terceira geração ou linguagens de alto nível**: essas linguagens se aproximam mais da linguagem humana, sendo mais fáceis de compreender. Os compiladores são os responsáveis pela tradução das instruções de alto nível em linguagem de máquina. Exemplos desse tipo de linguagem são: *Pascal, Modula, C* e *Ada*. Esta geração também inclui as linguagens orientadas a objetos como *C++*, *Smalltalk, Eiffel* e *Java*.
- **Linguagens de quarta geração**: são linguagens mais próximas da linguagem humana. Essas linguagens "caracterizam-se por apresentarem comandos capazes de realizar tarefas, que na terceira geração, implicariam a construção de programas detalhados" (AUDY, ANDRADE e CIDRAL, 2005, p. 174). As linguagens de definição e manipulação de dados, como SQL, *Progress* e *Informix*, são as principais representantes desta geração. Linguagens de quarta geração representam o mais alto nível de abstração das linguagens de programação.

2.11.1. *Java*

Java nasceu de uma iniciativa da *Sun Microsystems* que, em 1991, financiou um projeto interno com o codinome *Green*. Esse projeto resultou no desenvolvimento de uma linguagem de programação baseada em *C++*. Inicialmente essa linguagem foi batizada, pelo seu criador, *James Gosling*, de *Oak* (árvore de carvalho vista pela janela da *Sun*). No entanto, mais tarde descobriu-se que já havia uma linguagem de programação com esse nome. Existem muitas teorias a respeito de como a linguagem ganhou o nome *Java*. Uma delas diz que o nome *Java* foi sugerido por ser o nome da cidade de origem do café tomado pela equipe que trabalhava no projeto *Green*.

O alvo inicial do projeto *Green* era o mercado de dispositivos eletrônicos inteligentes, voltado ao consumo popular. Porém, esse mercado não se desenvolveu tanto quando a *Sun* imaginava, devido a isso, o projeto passou por algumas dificuldades.

Em 1993, a *World Wilde Web* se tornou muito popular, e "a equipe da *Sun* viu de imediato potencial de utilizar o *Java* para adicionar conteúdo dinâmico, como interatividade e animações, às páginas da *Web*" (DEITEL, 2005, p.6).

Em 1995, a *Sun* anunciou o *Java*, "não apenas como mais uma linguagem de programação, mas como uma nova plataforma de desenvolvimento" (FURGERI, 2002, p. 18). A partir daí *Java* se tornou muito popular e passou a ser utilizada para construção de aplicativos corporativos de grande porte, aplicativos para *desktops*, aplicativos para dispositivos móveis, aplicativos *web*, entre outros.

Alguns conceitos importantes relacionados à *Java*, definidos segundo Deitel (2005) são:

- **JVM**: a *Java Virtual Machine* é a responsável pela execução de programas escritos em *Java* em qualquer plataforma. Ela faz uma ponte entre o software e o sistema operacional. Dessa forma, um programa escrito em *Java* não tem acesso direto à máquina onde está sendo executado. A máquina virtual é uma camada extra entre a aplicação e o sistema operacional, responsável pela tradução do programa escrito em *Java* em linguagem específica da plataforma onde está sendo executado.
- **JDK**: o *Java Development Kit* contém as ferramentas básicas necessárias para desenvolver aplicativos em *Java*.
- **JRE**: o *Java Runtime Environment* é o ambiente de execução *Java*. Ele é composto pela JVM e pelas bibliotecas necessárias para executar aplicativos *Java*.
- ***Hotspot* e *Jit***: a JVM utiliza a tecnologias de *hotspots* para detectar pontos da aplicação que são mais utilizados e compilar esses pontos em linguagem nativa da plataforma. Dessa forma melhorando a performance da aplicação. O *Jit* (*Just in time compiler*) é o compilador utilizado pela JVM para transformar os *hotspots* em linguagem nativa.
- ***Bytecode***: é o código binário gerado pelo compilador *Java* (*javac*).
- ***ClassLoader***: é o responsável pela transferência dos *bytecodes* contidos nos arquivos *.class* para a memória.
- ***ByteCode Verifier***: avalia os *bycodes* carregados para memória para garantir que estes são válidos.

Segundo Deitel (2005), aplicativos construídos em *Java* passam por cinco fases:

1. **Edição**: refere-se à criação do programa em algum editor e ao armazenamento do arquivo com extensão *.java*;
2. **Compilação**: nesta fase o compilador transforma as instruções contidas no arquivo *.java* em *bytecodes* e os armazenas em arquivos com a extensão *.class*;
3. **Carga**: os *bytecodes* do arquivo *.class* são lidos por um carregador de classes e aloca-se espaço na memória para estes *bytecodes*;
4. **Verificação**: um verificador de *bytecodes* certifica-se de que todos os *bytecodes* são válidos e não violam restrições de segurança;
5. **Execução**: a JVM lê os *bytecodes* e os traduz para linguagem de máquina compreensível ao dispositivo.

Normalmente uma plataforma é composta por hardware e software. No entanto, a plataforma *Java* "difere da maioria das outras plataformas porque é composta apenas de um software operando sobre outra plataforma qualquer" (FURGERI, 2002, p. 22).

Geralmente, para que um programa seja executado em diferentes plataformas, é necessário que ele seja compilado na plataforma em que será executado. No caso da plataforma *Java*, um programa pode ser executado em várias plataformas sem que seja necessário compilar o código para cada uma delas.

A plataforma *Java* está divida em três edições:

- ***Java SE (Standard Edition)***: é a base para o desenvolvimento em *Java*. Contém todas as ferramentas necessárias para o desenvolvimento de aplicativos em *Java*. As duas outras edições da plataforma precisam dessa edição para serem executadas.
- ***Java EE (Enterprise Edition)***: voltada para o desenvolvimento de aplicativos de rede de grande porte, distribuídos e baseados na *web*.
- ***Java ME (Micro Edition)***: voltada ao desenvolvimento de aplicativos para dispositivos móveis, tais como PDAs, *smartphones*, celulares, etc.

2.11.2. *Flex*

O *Adobe Flex*, inicialmente desenvolvido pela *Macromedia* e posteriormente adquirido pela *Adode*, é uma tecnologia que permite o desenvolvimento de *Rich Internet Application*.

Aplicações desenvolvidas com *Flex* utilizam duas linguagens de programação:

- **MXML**: utilizada para desenhar a interface gráfica do sistema;

- ***ActionScript***: utilizado para customizar o comportamento de componentes desenvolvidos com MXML.

Linguagens de marcação baseadas em XML, como o MXML, não são ideais para implementar as operações que um sistema deve realizar quando, por exemplo, um usuário clicar em um botão. Para isso, é necessário utilizar linguagens como *ActionScript*, que permitem utilizar variáveis, laços de repetição, estruturas condicionais, etc.

O ponto central do *Flex* é o *Flex Software Development Kit* (*Flex* SDK, também conhecido como *Flex Framework*). Esse é o kit de desenvolvimento que oferece todas as funcionalidades necessárias para desenvolver aplicativos utilizando esta tecnologia. A partir da versão 3 do *Flex* SDK, este passou a ser *opensource*[13].

O *Flex Framework* contém um conjunto de classes que podem ser utilizadas para construção de aplicações *Flex*. De acordo com Kazoun e Lott (2007, p.1) o *framework Flex* "é escrito totalmente com classes em *ActionScript*, e define controles, *containers*, e gerenciadores de *design* para simplificar a construção de *Rich Internet Applications*".

O *Flex* define um modelo de desenvolvimento baseado em componentes. Junto com o SDK são disponibilizados uma série de componentes como botões, caixas de texto, *datagrids*, etc. No entanto, os desenvolvedores também podem criar seus próprios componentes.

Segundo Kazoun e Lott (2007), *Flex* é um conjunto de tecnologias com permitem desenvolver aplicações que serão executadas no *Flash Player*. As aplicações construídas em *Flex* utilizam o *Flex Framework* e são distribuídas utilizando o *Flash Player*. Essas aplicações contam com uma interface gráfica sofisticada e interativa.

O *Flex* pode ser integrado com outras linguagens de programação *server-side*[14], tais como *Java*, *Ruby*, linguagens da plataforma *Microsoft .Net*, etc.

2.12. Persistência de dados

O objetivo da persistência de dados é guardar os valores que estão sendo manipulados por um sistema, para que estes valores possam ser recuperados posteriormente. É possível salvar esses dados utilizando arquivos, porém, a abordagem mais comum é salvar os dados em bancos de dados.

[13] *Opensource* são projetos com distribuição livre, incluem não apenas os arquivos binários necessários para a execução do software, mas também o código fonte.

[14] Linguagens de programação *server-side* são utilizadas para realizar todo o processamento necessário no servidor onde a aplicação está hospedada.

Segundo Bauer e King (2006, p.8) em sistemas orientados a objetos, a persistência "permite a um objeto sobreviver ao processo que o criou. O estado do objeto pode ser guardado no disco, e um objeto com um mesmo estado pode ser recriado em algum ponto no futuro".

2.12.1. Banco de dados

Os bancos de dados tornaram-se muito importantes dentro das organizações. Bancos de dados são utilizados para armazenar as informações utilizadas pela empresa para realizar suas atividades. As empresas utilizam os bancos de dados para manter essas informações atualizadas e exatas.

Segundo Harrington (2002, p.3),

> Sem esses repositórios de dados de missão crítica, a maioria das empresas é incapaz de realizar suas transações diárias normais, muito menos de criar relatórios de resumo que ajudam o gerente a tomar decisões corporativas estratégicas. Para serem úteis, os dados em um banco de dados, devem ser exatos, completos e organizados de tal maneira que os dados possam ser recuperados quando necessário e no formato requerido.

Date (2000, p.36), define banco de dados como

> Uma coleção de dados organizados e integrados que constitui uma representação natural de dados, sem imposição de restrições ou modificações para ser adequada a computadores e que possa ser utilizada por todas as aplicações relevantes, sem duplicação de dados.

Os bancos de dados utilizam um modelo de dados. Esse modelo representa a forma como os dados são armazenados e como eles se relacionam. Alguns modelos de dados são:

- Hierárquico;
- Em redes;
- Relacional;
- Orientado a objetos.

Atualmente o modelo relacional é o mais utilizado, esse modelo foi o primeiro a ser utilizado para aplicações comerciais. Nesse modelo os dados são organizados em tabelas. Cada tabela possui um nome único e um conjunto de colunas que representam os atributos das entidades.

Sistemas de banco dados disponibilizam dois tipos de linguagem para definição e manipulação de dados:

- **Linguagem de definição de dados** (*Data-Definition Language* – DDL): são utilizadas para definir as tabelas utilizadas pelo banco de dados;
- **Linguagem de manipulação de dados** (*Data-Manipulation Language* – DML): são utilizadas para consultar, atualizar, inserir ou remover dados de um banco de dados.

Para manter a consistência dos dados armazenados nos bancos de dados é utilizada uma técnica conhecida como normalização de dados. Essa técnica consiste numa seqüência de passos que definem quais são as entidades e seus relacionamentos e atributos. A normalização de dados é dividida em formas normais:

- **1ª Forma Normal**: atributos multivalorados são removidos da tabela e passa-se a utilizar um atributo chave para fazer o relacionamento e manipulação dos dados da tabela original.
- **2ª Forma Normal**: todos os atributos que não são chaves devem depender diretamente da chave primária da tabela. Os atributos que não possuem dependência direta devem ser removidos.
- **3ª Forma Normal**: são removidos todos os atributos que dependem de outros atributos que não são chave da tabela.

2.12.1.1. Sistemas gerenciadores de banco de dados

Os Sistemas Gerenciadores de Bancos de dados (SGBD) fornecem um conjunto de ferramentas que facilitam a manipulação das informações contidas em um banco de dados. Os SGBDs representam uma camada entre os dados físicos e as aplicações ou usuários que utilizam esses dados.

Kort, Silberschatz e Sudarshan (1991, p.1) afirmam que o principal objetivo dos SGBDs é "proporcionar um ambiente tanto conveniente quanto eficiente para a recuperação e armazenamento das informações do banco de dados".

Pelo fato de gerenciarem grandes volumes de dados, os SGBDs devem definir estruturas de armazenamento e mecanismos adequados para manipular informações. Além disso, os SGBDs também precisam fornecer mecanismos de segurança para as informações armazenadas nos bancos de dados.

De acordo com Bauer e King (2006, p. 8),

> Bancos de dados relacionais modernos fornecem uma representação estruturada dos dados persistentes, permitindo a manipulação, a classificação, a busca e a agregação dos dados. Os sistemas de gerenciamento de banco de dados são responsáveis por

> gerenciar a concorrência e a integridade dos dados; eles são responsáveis por compartilhar os dados por entre os múltiplos usuários e múltiplas aplicações. Eles garantem a integridade dos dados através de regras de integridade que são implementadas com restrições (*constraints*). Um sistema de gerenciamento de banco de dados fornece segurança no nível dos dados.

2.12.1.2. MER (Modelo entidade-relacionamento)

O Modelo Entidade-Relacionamento (MER) foi criado para ser compreendido pelos usuários. Por isso, o MER não se preocupa, por exemplo, com a forma como os dados são armazenados fisicamente. O MER é utilizado durante a fase de projeto para modelar bancos de dados.

De acordo com Korth, Silberschatz e Sudarshan (1999, p. 21),

> O modelo entidade-relacionamento (E-R) tem por base a percepção de que o mundo real é formado por um conjunto de objetos chamados entidades e pelo conjunto de relacionamentos entre esses objetos. Foi desenvolvido para facilitar o projeto de bancos de dados, permitindo a especificação do esquema da empresa, que representa toda a estrutura lógica do banco de dados. O modelo E-R é um dos modelos com maior capacidade semântica; os aspectos semânticos do modelo se referem à tentativa de representar o significado dos dados.

O MER utiliza três conceitos básicos:

- **Conjuntos de entidades**: as entidades representam algo que existe no mundo real e que pode ser identificada de forma unívoca em relação a outros objetos. Um conjunto de entidades representa todas as entidades do mesmo tipo que compartilham as mesmas propriedades.
- **Conjuntos de relacionamentos**: um relacionamento representa uma associação existente entre uma ou várias entidades. Um conjunto de relacionamentos representa relacionamentos do mesmo tipo.
- **Atributos**: representam as propriedades que uma entidade possui.

2.12.2. Mapeamento objeto-relacional

Praticamente todo sistema comercial guarda seus dados em bancos de dados relacionais. Porém, hoje, grande parte das linguagens de programação utilizadas para desenvolver esse tipo de sistema são orientadas a objetos. Para tentar minimizar os problemas decorrentes da utilização de linguagens orientadas a objetos com bancos de dados relacionais, surgiu o mapeamento objeto-relacional.

O Mapeamento Objeto-Relacional (*Object-Relational Mapping* – ORM) representa o conjunto de técnicas e ferramentas utilizadas para fazer o mapeamento dos objetos de aplicações Orientadas a Objetos para entidades dos bancos de dados relacionais. Ou, nas palavras de Bauer e King (2006) "mapeamento objeto/relacional é a persistência automatizada (e transparente) dos objetos em uma aplicação [...] para as tabelas de um banco de dados relacional".

Existem diversos *frameworks* no mercado que dispõem das funcionalidades necessárias para a implementação do mapeamento objeto relacional de forma automatizada. Segundo Bauer e King (2006), alguns benefícios da utilização de *frameworks* ORM são:

- **Produtividade**: a utilização de um *framework* ORM permite reduzir o tempo de desenvolvimento da camada de persistência. Isso ocorre porque esses *frameworks* disponibilizam as funcionalidades necessárias para mapear e manipular os objetos de uma aplicação de forma transparente.
- **Manutenibilidade**: além de diminuir o número de linhas de código e, conseqüentemente, tornar o software mais legível, o ORM permite isolar o modelo de objetos do modelo relacional. Ao fazer isso, o ORM impede que pequenas alterações em um modelo afetem o outro.
- **Independência de fornecedor**: *frameworks* ORM abstraem da aplicação a necessidade de interagir diretamente com o banco. Dessa forma, o sistema se torna independente do banco de dados utilizado, aumentando a portabilidade do sistema.

Fussel (1997 *apud* BAUER e KING, 2006), define quatro níveis de qualidade ORM:

- **Relacional puro** (*Pure Relational*): todo o sistema é desenvolvido com base no modelo relacional e operações usando SQL [15]. Esse nível de ORM atende as necessidades de pequenas aplicações. Porém, para sistemas de grande porte, onde aspectos como portabilidade e manutenibilidade são importantes, essa abordagem torna-se deficiente.
- **Mapeamento leve** (*Light Object Mapping*): as entidades são representadas por classes que são mapeadas manualmente para as tabelas do banco relacional. O SQL é escrito manualmente. Essa abordagem é muito comum e, para aplicações com poucas entidades, apresenta resultados satisfatórios.
- **Mapeamento médio** (*Médium Object Mapping*): o sistema é construído usando um modelo de objetos. O código SQL é gerado em tempo de execução através da

[15] *Structured Query Language* (Linguagem de Consulta Estrutura). É a linguagem de definição e manipulação de dados padrão para a maioria dos bancos de dados relacionais.

utilização de *frameworks* ORM.

- **Mapeamento completo** (*Full Object Mapping*): esse nível de mapeamento suporta composição, herança, polimorfismo e persistência de transitividades. A persistência ocorre de forma transparente.

3. DESCRIÇÃO DAS ATIVIDADES PRÁTICAS

Este capítulo descreve as atividades realizadas durante o desenvolvimento do *framework* e os artefatos produzidos em cada atividade. É neste momento que se põem em prática os conceitos definidos na fundamentação teórica deste projeto. O objetivo é desenvolver um produto de qualidade que segue os princípios propostos pela engenharia software.

3.1. Local do desenvolvimento

O *framework* foi projetado para atender as necessidades da empresa *Clean* Informática Ltda. Essa empresa atual no mercado de desenvolvimento e comercialização de sistemas desde 1994. O foco principal da empresa é o desenvolvimento de software para empresas de prestação de serviços. O principal produto oferecido pela empresa é o *Simps* (Sistema Integrado Multimarcas de Prestação de Serviços).

O *Simps* foi desenvolvido para atender empresas de médio e grande porte que atuam na área de prestação de serviços e venda de peças e produtos acabados. Ele é composto por diversos módulos que visam atender a todas as necessidades das prestadoras de serviços. Entre esses módulos estão:

- ***Simps* Revenda**: trata as rotinas referentes aos setores: comercial, estoque e financeiro;
- ***Simps* CRM**[16]: atua na gestão da área comercial, envolve a administração de contatos, captação de novos negócios, gerenciamento dos históricos, mala direta, propostas, relatórios e índices gerenciais de acompanhamento por vendedores e formas de contatos;
- ***Simps* Financerio**: é um modulo independente que visa gerenciar o setor financeiro de uma empresa;
- **Simps PDV**: responsável pela emissão dos documentos fiscais: cupons, notas fiscais e controle de caixa;
- ***Simps* TEF**: trabalha integrado ao *Simps* PDV, é o responsável pela comercialização com cartões de crédito ou débito;

[16] *Customer Relationship Management*. São ferramentas que disponibilizam uma série de funcionalidades de apoio a vendas, marketing e serviços aos clientes.

Além do *Simps*, a empresa também possui outros sistemas, entre eles pode-se citar o *Service Master* (SVM) e *Agines Network*. O *SVN* surgiu para integrar, padronizar e automatizar os processos existentes entre fábricas e redes de serviços autorizados. O *SVN* pode ser integrado ao *Simps* ou ao ERP *(Enterprise Resource Planning*) utilizado pela fábrica ou pela prestadora de serviços.

O *Agines Network* é um sistema desenvolvido para gestão empresarial. Esse sistema visa atender a diversos segmentos, entre eles: indústria, comércio, ferramentaria e marketing multinível.

3.1.1. Situação atual

Cada sistema desenvolvido pela empresa foi construído utilizando uma linguagem de programação diferente. As linguagens utilizadas hoje são *Delphi* (*Simps*), *PHP* (*Service Master*), *Java* (*Agines Network*) e, em projetos mais recentes, as linguagens da plataforma *Microsoft .Net* e *Flex*.

Nos últimos anos a empresa vem tentando minimizar o esforço de desenvolvimento ao projetar soluções que possam ser reutilizadas. No entanto, isso se torna difícil pelo fato de cada sistema construído pela empresa utilizar uma linguagem de programação diferente. Outro fator que agrava essa situação, é o fato de que muitos dos sistemas já existentes não foram projetados visando à reutilização de componentes.

Um dos problemas encontrado hoje é o fato de todos os sistemas terem que implementar rotinas que são comuns a todos os sistemas. Um exemplo recente disso é a construção do módulo financeiro para o *Agines Network*. Muito do que será implementado no *Agines Network* já existe no *Simps*, porém, a estrutura dos dois sistemas dificulta a integração para utilização de um módulo inteiro. Seria necessário realizar modificações drásticas na estrutura dos dois sistemas para que eles comportassem essa integração. E, além disso, os clientes precisariam adquirir os dois sistemas para utilizar o módulo financeiro.

A única forma de integrar os sistemas existentes é através de agentes[17] que transferem dados de uma aplicação para outra. Um exemplo disso é a integração entre *Agines Network* e *Simps PDV* para emissão de cupom fiscal (descrita na seção de cenários arquiteturais mais adiante). Outra situação onde ocorre esse tipo de integração é quanto é necessário que os

[17] Os agentes representam pequenos aplicativos que coletam dados do banco de dados de uma aplicação e inserem no banco de dados de outra aplicação. Esses aplicativos também são os responsáveis por todas as validações necessárias para realizar a integração entre os sistemas.

sistemas desenvolvidos pela empresa trabalhem em conjunto com os sistemas já utilizados pelos clientes.

Apesar dos problemas, existe um lado bom. A empresa percebeu que precisa desenvolver soluções que possam ser reutilizadas em diversos sistemas. Devido a isso a empresa passou a investir em projetos reutilizáveis e pretende de construir um *framework* que centralize os componentes que podem ser reutilizados em diversos projetos.

3.1.2. *Agines Network*

Agines Network é um ERP voltado a diversos setores, como comércio, indústria e ferramentaria. Ele é dividido em módulos e, alguns desses módulos podem ser comercializados independentemente. Alguns módulos do sistema são: produção, compras, vendas, estoque, fiscal, etc.

O sistema foi adquirido pela empresa *Clean* Informática Ltda, em abril de 2006. Nessa época, o sistema já possuía uma infra-estrutura precária e contava com os seguintes módulos: Cadastros, Configurações, Estoque, Ordem de Serviço, Compras e Nota Fiscal. Os dois últimos módulos estavam incompletos e foram melhorados pela empresa. Após adquirir o sistema, os desenvolvedores e analistas tiveram em torno de dois meses para conhecer o sistema antes de iniciar o desenvolvimento e a manutenção.

O sistema foi desenvolvido utilizando *applets*[18] *Java* e banco de dados *MySQL*. A tecnologia utilizada pelos *applets* apresenta algumas vantagens. A principal é o fato de que aplicativos escritos em *applets* podem ser executados em qualquer navegador *web*, desde que o navegador possua o *Java Plugin*[19] instalado. No entanto, os *applets* também apresentam uma grande desvantagem: embora eles sejam armazenados em um servidor, eles precisam ser baixados para a máquina do cliente para serem executados.

Quando os *applets* são utilizados para aplicativos de pequeno porte, o fato de precisar baixar para a máquina do cliente não é um problema. Mas, no caso de um ERP, isso se torna um grande empecilho. Os *applets* limitam a utilização de *frameworks* e APIs, pois estes também precisarão ser baixados para a máquina do cliente. Dessa forma, muitas funcionalidades que estão presentes em *frameworks* e APIs disponíveis no mercado precisam

[18] *Applets* são aplicativos escritos em *Java* que podem ser executados dentro de um navegador *web*. Ao acessar uma página que contenha *applets*, estes são automaticamente baixados para a máquina do cliente e executados.

[19] *Plugin* que adiciona a capacidade de executar aplicativos *Java* em um navegador *web*.

ser reimplementadas no sistema. O que torna os *applets* limitados, na verdade, é a velocidade e a qualidade da conexão com a Internet, embora esta tenha melhorado bastante nos últimos anos, ela ainda é deficiente.

Porém, o principal problema do sistema hoje é a má estruturação do mesmo. Regras de negócio, acesso a dados e interface gráfica estão extremamente inter-relacionados. Ou seja, em uma única classe *Java* está presente o código referente às regras de negócio, ao acesso e manipulação de dados e a interface gráfica do sistema. Qualquer alteração necessária (seja relacionada a regras de negócio ou a manipulação de dados, ou a interface gráfica) irá impactar diretamente nas outras "*camadas*".

Um grande problema causado pela má estruturação do sistema é a duplicação de código. Existem rotinas no sistema, como, por exemplo, a movimentação de estoque, que são implementadas exatamente da mesma forma em diversas partes do sistema. Isso ocorre porque o sistema trabalha diretamente com o banco de dados, então, se em uma tela são utilizados dados de uma determinada tabela para movimentar o estoque, em outra tela os dados vêm de outra tabela.

Outro problema enfrentado pelo sistema diz respeito à dependência do banco de dados. Todas as instruções necessárias para a consulta ou manipulação de dados são codificadas manualmente em instruções SQL. O banco utilizado pelo sistema disponibiliza um conjunto de funções que facilitam essa tarefa. No entanto, nem todas as funções disponíveis no *MySQL* estão disponíveis em outros bancos. Além disso, o sistema não possui uma camada de persistência que abstraia os detalhes de cada banco de dados. Portanto, se fosse necessário que o sistema passasse a utilizar outro banco de dados, seria necessário reescrever todas as consultas SQL disponíveis no sistema.

3.1.3. Problemas do *framework* atual

O *framework* utilizado pelo *Agines Network*, nada mais é do que uma biblioteca de classes que disponibiliza um conjunto de componentes gráficos e algumas funções para validação e formatação de dados e controles de tela. O *framework* não determinada a arquitetura do sistema.

O *framework* atual é utilizado principalmente para construir a interface gráfica do sistema, pois, dispõem de vários componentes gráficos que encapsulam alguns dos controles utilizados pelo sistema. No entanto, esses componentes apresentam alguns problemas, o

principal deles é a duplicação de código. Muitas funcionalidades, que são utilizadas por diversos componentes, e que poderiam estar em uma classe à parte, são repetidas dentro de cada componente. Além de dificultar a manutenção do *framework*, muitas dessas funcionalidades não são utilizadas pelo componente.

3.2. Escopo do projeto

Um *framework* encontra-se em constante evolução. É muito difícil determinar tudo que um *framework* deve possuir de uma única vez. Isso ocorre, principalmente, devido às constantes mudanças nos requisitos dos sistemas que dão origem a um *framework*.

Devido a isso, inicialmente o *framework* proposto por este projeto contemplará apenas os componentes essenciais para a construção de sistemas. Ou seja, será construída a base para o desenvolvimento de sistemas, utilizando uma estrutura flexível e padrões arquiteturais que atendam aos requisitos dos sistemas que serão construídos a partir do *framework*.

Os componentes que serão desenvolvidos incluem componentes para acesso e manipulação de dados, componentes para construção de interfaces gráficas e integração entre sistemas. Além de componentes mais específicos, como perfil de usuário, envio de e-mail e controle de acesso.

Para construir o *framework* toma-se como base a solução dos problemas encontrados no sistema *Agines Network*, bem como os requisitos dos outros sistemas desenvolvidos pela empresa.

3.3. Levantamento de requisitos

O levantamento de requisitos do *framework* iniciou-se com a identificação de cenários arquiteturais. Através desses cenários foram identificados os atores e casos de uso do *framework*. Após a identificação dos casos de uso, estes foram priorizados e detalhados.

O objetivo do levantamento de requisitos é identificar quais são as carências dos sistemas desenvolvidos pela empresa. Através de reuniões com os analistas de sistemas da empresa e da análise dos sistemas existentes, foi possível identificar pontos em comum entre os sistemas e determinar os requisitos iniciais do *framework*.

3.3.1. Cenários arquiteturais

Os cenários arquiteturais ajudam a identificar casos de uso, requisitos funcionais e não-funcionais do sistema, arquiteturas que podem atender as necessidades do *framework* etc. A Tabela 12 descreve o modelo que será utilizado para descrever os cenários arquiteturais identificados.

Tabela 12 – Modelo de descrição de cenário arquitetural

Identificador	Nome utilizado para identificar o cenário arquitetural. Deve ser breve e objetivo, indicando do que trata o cenário em questão.
Cenário	Descrição do objetivo principal do cenário e como ocorre (ou deveria ocorrer) a interação entre atores e sistemas.
Problemas	**Perspectiva do usuário**
	Descreve os problemas encontrados pelos usuários para interagir com o sistema no cenário em questão.
	Perspectiva do desenvolvimento de software
	Descreve os problemas encontrados para transformar o cenário em um software.
Solução proposta	Descreve a solução proposta para implementar o cenário em questão.
Requisitos arquiteturais	Descreve os principais requisitos arquiteturais (ou não-funcionais) do cenário em questão.
Arquiteturas candidatas	Descreve as possíveis soluções que podem ser utilizada para implementar o cenário em questão.

Fonte: Kamyla Estuqui Parrado (2008)

Alguns cenários descrevem situações reais encontradas nos sistemas desenvolvidos pela empresa. Outros, no entanto, apresentam um cenário genérico que pode ser encontrado em qualquer aplicação comercial.

Para auxiliar a identificação e análise dos cenários arquiteturais foi utilizado o método de análise arquitetural SAAM. Para analisar os benefícios e problemas da solução escolhida foi utilizada a ferramenta SWOT (***S**trengths*, ***W**eaknesses*, ***O**pportunities*, ***T**hreats* – Forças, Fraquezas, Oportunidades, Ameaças). Essa ferramenta costuma ser utilizada para avaliação de cenários. O SWOT permite identificar forças e fraquezas, que dizem respeito ao ambiente

interno do cenário, e oportunidades e ameaças, que se referem ao ambiente externo ao cenário.

Além dos itens apresentados na Tabela 12, cada cenário apresenta uma breve descrição das alternativas que podem ser utilizadas para desenvolver a solução.

A Tabela 13 descreve um dos cenários arquiteturais mais importantes para os sistemas desenvolvidos pela empresa hoje: a manipulação de dados. Esse cenário trata das operações necessárias para definir e manipular dados utilizando banco de dados relacionais. Um dos objetivos desse cenário é identificar soluções para os problemas decorrentes da utilização de bancos de dados relacionais em aplicações orientadas a objeto.

Tabela 13 – Cenário arquitetural: Manipulação de dados

Identificador	Manipulação de dados
Cenário	Sistemas de informação precisam armazenar os dados utilizados, produzidos e/ou recebidos em tempo de execução em algum meio físico. Além de armazenar os dados, esses sistemas precisam manipular os dados de alguma maneira. Essa manipulação, normalmente ocorre através de quatro operações básicas conhecidas como CRUD (*create*, *read*, *update*, *delete* – criar, ler, atualizar, deletar). As aplicações podem receber dados através de uma interface gráfica ou da integração com outros sistemas. Os usuários preenchem os campos disponíveis na interface e pressionam um botão (OK, Gravar, Salvar, etc). Os dados são validados (de acordo com as regras de negócio da aplicação) e, se o resultado dessa validação for positivo, então as informações são enviadas para um banco de dados. Senão, o usuário recebe algum tipo de mensagem informando qual erro ocorreu. Além disso, através dos dados informados pelos usuários, ou de alguma outra fonte, o sistema realiza algum tipo de processamento e retorna esses dados para o usuário e grava os dados no banco.
Problemas	**Perspectiva do usuário**
	1. Os dados normalmente são os bens mais preciosos de uma organização. Ao interagir com um sistema, os usuários precisam ter certeza de que as informações que estão sendo informadas serão armazenadas e tratadas de forma segura. 2. Existem consultas complexas feitas ao banco de dados que podem

	comprometer o desempenho de um sistema. Isso afeta diretamente a usabilidade do sistema, pois, sistemas lentos, tendem a ser menos eficientes do que sistemas com alto desempenho. 3. Nem todos os sistemas dão suporte à integridade de dados. Por exemplo, se, durante a execução de uma transação, a energia acabar, todos os dados podem ser perdidos ou corrompidos, pois nem todos os sistemas implementam os mecanismos necessários para garantir a integridade dos dados.
	Perspectiva do desenvolvimento de software
	1. Bancos de dados relacionais utilizam SQL para manipulação de dados. Embora exista um padrão denominado SQL 92 que determina certas características que todos os SGBD devem possuir, cada SGBD tem liberdade para implementar funcionalidades específicas que facilitam a manipulação de dados. Porém, quando um sistema utiliza essas funcionalidades, ele se torna dependente do banco de dados, dificultando a portabilidade do sistema. 2. Ao codificar o SQL manualmente em aplicações orientadas a objetos, toda a aplicação passa a depender do modelo relacional. Qualquer modificação no modelo relacional deverá ser codificada manualmente na aplicação. Isso envolve alterar todas as classes que acessem uma determinada tabela do banco de dados. As regras de negócio devem ser modificadas para amparar as alterações do modelo relacional. Erros podem ser introduzidos no sistema, devido à grande quantidade de modificações necessárias. E, provavelmente, todo o sistema deverá ser testado novamente. 3. A codificação manual da camada de persistência de dados, utilizando SQL, pode despender muito mais tempo de desenvolvimento e provocar muito mais erros do que a utilização de mecanismos de persistência automatizados. 4. Quando dados são manipulados em bancos de dados, é necessário que exista um controle sobre as transações[20] realizadas, pois, caso

[20] No contexto dos bancos de dados, uma transação representa uma operação, normalmente de inserção, alteração ou exclusão de dados, que só pode ser executada se não ocorrer nenhum tipo de erro. Caso alguma parte da transação falhe, todas as outras devem ser desfeitas para garantir a consistência/integridade dos dados.

	ocorra algum tipo de erro, essa transação precisa ser desfeita para garantir a integridade dos dados. O gerenciamento manual dessas transações pode despender muito tempo de desenvolvimento e é muito propício a erros. 5. Realizar o mapeamento entre as classes de uma aplicação orientada a objeto e as tabelas de um banco de dados relacional manualmente é uma tarefa difícil e propícia a erros.
Solução proposta	Construção de uma camada de persistência que: 1. Disponibilize as operações básicas para manipulação de dados: inclusão, atualização, consulta e remoção; 2. Seja independente do banco de dados utilizado; 3. Realize o mapeamento objeto-relacional das classes do sistema para tabelas do banco de dados; 4. Forneça mecanismos de extensão que permitam personalizar as operações básicas fornecidas, para atender necessidades especificas de cada sistema; 5. Gerencie as transações realizadas no banco de dados; 6. Permita o *log* e *trace* de erros que possam ocorrer durante a execução.
Requisitos arquiteturais	▪ **Desempenho**: bancos de dados são o centro de qualquer sistemas de informação. Portanto, é imprescindível que a manipulação desses dados ocorra da forma ágil. ▪ **Flexibilidade**: a camada de persistência deve ser flexível o suficiente para suportar diversos tipos de bancos de dados. E, além de oferecer as funcionalidades de um CRUD, permitir a criação de novas funcionalidades. ▪ **Portabilidade**: os recursos existentes na camada de acesso a banco de dados devem ser portáveis a qualquer SGBD. Ou seja, as funcionalidades implementadas na camada de persistência devem ser suportadas pelos principais SGBDs do mercado. ▪ **Segurança**: pelo fato dos bancos de dados serem o centro dos sistemas de informação, eles acabam armazenando informações de suma importância para a organização. Muitas vezes, essas

	informações são confidenciais, e o acesso irrestrito a elas pode causar sérios danos à organização. Devido a isso, a segurança se torna indispensável a qualquer camada de acesso a banco de dados.
Arquiteturas candidatas	▪ Desenvolvimento proprietário de um *framework* de persistência; ▪ Utilizar *frameworks* de persistências existentes no mercado para construir uma camada de persistência.

Fonte: Kamyla Estuqui Parrado (2008)

Foram identificadas duas alternativas para o desenvolvimento da camada de persistência do *framework*. A primeira envolve o desenvolvimento de um *framework* de persistência proprietário. Ou seja, como alguns desenvolvedores gostam de dizer: "reinventar a roda". Porém, isso iria contra o principio de reutilização de componentes de software proposto pela engenharia de software. Dessa forma, optou-se pela segunda alternativa, que consiste no desenvolvimento de uma camada de persistência utilizando *frameworks* ORM disponíveis no mercado.

Quando se trata de *frameworks* escritos para a linguagem de programação *Java*, existem diversas opções. Para determinar qual *framework* é o mais adequado as necessidades da empresa foram feitas pesquisas sobre as principais ferramentas para persistência de dados disponíveis no mercado. Algumas dessas ferramentas são:

- ***Hibernate***: é um dos *frameworks* ORM mais populares do mercado. O *Hibernate* segue a especificação da JPA[21]. Este *framework* permite o mapeamento das classes em *Java* tanto através de arquivos de configuração escritos em XML quanto de *Annotations*[22]. O *Hibernate* é capaz de trabalhar com a maioria dos bancos de dados relacionais disponíveis no mercado, desde que estes possuam um *driver* JDBC[23] e o dialeto utilizado pelo banco seja suportado pelo *Hibernate*. O *Hibernate* fornece mecanismos para a criação, atualização e exclusão de objetos que são totalmente transparentes ao desenvolvedor. Além disso, o *Hibernate* permite consultar os objetos

[21] JPA, ou *Java Persistence API*, é a API (*Application Program Interface*) de persistência de dados do *Java*. Foi desenvolvida para facilitar o mapeamento objeto-relacional permitindo o mapeamento das classes através de *Annotations*, ao invés de utilizar extensos e complexos arquivos XML. A utilização de *frameworks* baseados na especificação da JPA garante que as aplicações desenvolvidas não serão dependentes do fornecedor do *framework*.

[22] *Annotations* (Anotações) são meta-informações que não interferem diretamente sobre a linguagem de programação. As anotações são utilizadas por ferramentas ou bibliotecas para tratar o código *anotado* de uma forma diferente da qual o código seria tratado se não estivesse sido anotado.

[23] *Java DataBase Connectivity*. Uma *API* escrita em *Java* que permite acesso a bancos de dados.

armazenados no banco através de HQL[24] ou da API *Criteria* (HIBERNATE, 2008). Por ser um padrão de mercado, o *framework* possui uma ampla documentação, tanto on-line quanto em livros.

- ***TopLink***: é um *framework* para persistência de dados em *Java*. Inicialmente o *framework* foi desenvolvido em *Smalltalk* e, posteriormente, uma versão em *Java* foi adicionada a linha de produtos. Após ter adquirido o *framework*, a empresa *Oracle* doou o código fonte para o projeto *Sun Microsystems java.net Glassfish*. Isso resultou no desenvolvimento do *TopLink Essentials* (versão *open-source* do produto *Oracle TopLink*), uma implementação de referência da JPA. O *TopLink* oferece suporte ao desenvolvimento de consultas utilizando QBE (*Query by Example*), EJB QL[25] e SQL. Esse *framework* também permite a utilização de *stored procedures* (ORACLE, 2008).
- ***iBATIS***: é um dos *frameworks* ORM mais simples de utilizar. O *iBATIS* permite unir objetos, consultas SQL e *stored procedures* através de um descritor XML (arquivo de configuração). O *iBATIS* é composto pelo *iBATIS Data Mapper* e pelo *iBATIS Data Access Objects*. O *iBATIS Data Mapper* é o responsável pelo mapeamento da classes *Java* em tabelas do banco de dados. O *iBATIS Data Access Objects* é o responsável pelo acesso aos dados armazenados (IBATIS, 2008).
- ***Prevayler***: este *framework* difere dos anteriores no sentido de não utilizar bancos de dados e SQL. Ele trabalha apenas com objetos e garante que esses objetos sempre estejam em memória. Devido a isso, o desempenho desse *framework* é muito superior aos *frameworks* de persistência ORM tradicionais. Por não trabalhar com bancos de dados este *framework* não apresenta características como controle de acesso, integridade, consultas SQL, etc. Ao utilizar este *framework*, toda a lógica necessária para acessar objetos precisa ser desenvolvida manualmente pelo programador. Isso pode se tornar um problema quanto for necessário utilizar consultas complexas para acessar os dados. Embora o *framework* não ofereça suporte direto ou uso de SQL, é possível utilizá-lo em conjunto com outros *frameworks* que permitem realizar consultas SQL em objetos. Um desses *frameworks* é o *JoSQL*, que permite realizar consultas SQL em instâncias de classes *Java*. O *Prevayler* e o *JoSQL* podem ser utilizados em conjunto para minimizar o esforço de desenvolvimento de consultas complexas de acesso à dados (PREVAYLER, 2008).

[24] *Hibernate Query Language*. É um dialeto SQL utilizado pelo *Hibernate*. Embora sua sintaxe se pareça com SQL, o HQL é totalmente orientado a objetos.

[25] Linguagem de consulta semelhante ao SQL, porém adaptada para trabalhar com objetos *Java*, ao invés de um esquema relacional.

Através da análise realizada sobre os principais *frameworks* de persistência de dados do mercado, optou-se pela utilização do *Hibernate* devido, principalmente, a ampla aceitação que este tem no mercado. A Tabela 14 apresenta a análise SWOT realizada sobre o *framework Hibernate*.

Tabela 14 – Análise SWOT do *framework Hibernate*

Forças	**Fraquezas**
1. Permite mapear as classes *Java* através da utilização de *Annotations* e POJOs (*Plain Old Java Objects*); 2. Segue a especificação da JPA; 3. Bibliografia e documentação sobre a tecnologia.	1. Não trabalha adequadamente com *stored procedures*; 2. Depende do *driver* JDBC disponibilizado pela empresa desenvolvedora do SGBD.
Oportunidades	**Ameaças**
1. Comunidade de usuários muito grande; 2. Ampla aceitação do mercado; 3. *Framework* padrão para desenvolvimento de camadas de persistência.	1. O projeto pode ser descontinuado; 2. Versões futuras podem ser incompatíveis com os produtos desenvolvidos utilizando versões anteriores.

Fonte: Kamyla Estuqui Parrado (2008).

A Tabela 15 descreve o cenário arquitetural de interface gráfica. Por ser o principal meio através do qual o usuário interage com as aplicações, a interface gráfica de um sistema deve ser bem projetada, caso contrário, a usabilidade do sistema pode ser comprometida. O objetivo desse cenário é identificar alternativas para a construção de interfaces gráficas seguindo o paradigma da *Web* 2.0 e das *Rich Internet Applications*.

Tabela 15 – Cenário arquitetural: Interface gráfica com o usuário

Identificador	Interface gráfica com o usuário
Cenário	Os usuários interagem com o sistema através de interfaces gráficas que fornecem acesso aos serviços oferecidos por um sistema. Essa interação ocorre quando o usuário inicia o sistema, navega entre as telas disponíveis e visualiza, insere, edita ou remove as informações disponíveis.
Problemas	**Perspectiva do usuário**
	1. Falta de padronização entre as telas do sistema;

	2. Falta de usabilidade: 2.1. Telas muito complexas; 2.2. Telas com excesso de informações; 2.3. Telas de difícil entendimento.
	Perspectiva do desenvolvimento de software
	1. Duplicação de código para implementar em cada tela funcionalidades que poderiam ser encapsuladas em componentes; 2. Bibliotecas de componentes gráficos padrão disponíveis não atendem as necessidades do sistema; 3. Algumas tecnologias disponíveis para o desenvolvimento de interfaces gráficas apresentam incompatibilidade entre navegadores *web*. Devido a isso, é necessário escrever o mesmo código várias vezes, porém adaptando-o às necessidades do navegador onde o sistema poderá ser executado.
Solução proposta	Construção de componentes gráficos que encapsulem funcionalidades como controle e validação de campos e forneçam mecanismos padrão para o desenvolvimento de interfaces gráficas.
Requisitos arquiteturais	▪ **Usabilidade**: este é um dos requisitos arquiteturais mais importantes quando se trata de interfaces gráficas com o usuário. É através dessas interfaces que o usuário interage com o sistema, portanto, a interface gráfica precisa ser fácil de usar e agradável ao usuário. ▪ **Desempenho**: este requisito arquitetural está diretamente associado à usabilidade das telas do sistema. Se a interface gráfica não for bem projetada ela pode dificultar a utilização do sistema. Se a interface, por exemplo, possuir muitos componentes gráficos, ou muitos efeitos, a performance da tela pode ser comprometida.
Arquiteturas candidatas	▪ Utilização de componentes gráficos já existentes; ▪ Desenvolvimento de componentes gráficos personalizados através da utilização de componentes gráficos já existentes utilizando linguagens de programação para desenvolvimento de RIAs.

Fonte: Kamyla Estuqui Parrado (2008)

A utilização de componentes gráficos já existentes implica na necessidade de implementar manualmente em cada tela, funcionalidades básicas que poderiam ser implementadas dentro dos componentes. No entanto, é possível utilizar estes componentes para desenvolver componentes gráficos personalizados que atendam as necessidades do *framework*. Para isso, no entanto, será utilizada uma linguagem de programação que permite o desenvolvimento de *Rich Internet Applications*. As principais representantes dessas linguagens são:

- ***OpenLaszlo***: é uma plataforma, *opensource*, para desenvolvimento de RIAs. A plataforma consiste em um servidor (*OpenLaszlo Server*) e uma linguagem de programação (LZX, semelhante ao MXML do *Adobe Flex*). O *OpenLaszlo Server* é o responsável por compilar as aplicações escritas em LZX, transformando-as em arquivos binários executáveis. A principal desvantagem decorrente da utilização do *OpenLaszlo*, é o fato da plataforma não dispor de uma IDE [26] que auxilie o desenvolvimento de aplicações (OPEN LASZLO, 2008).
- ***Microsoft Silverlight***: é um *plugin* [27] para navegadores *web* que permite o desenvolvimento e visualização de *Rich Internet Applications*. A principal desvantagem dessa ferramenta é o fato de ser uma tecnologia proprietária, ou seja, é uma tecnologia paga. *Silverlight* é capaz de se integrar com diversas linguagens de programação, entre elas *Visual Basic*, *C#*, *Python* e *Ruby* (MICROSOFT, 2008).
- ***Adobe Flex***: é uma plataforma para desenvolvimento de RIAs. É capaz de trabalhar em conjunto com diversas linguagens de programação *server side*, como *Java*, *Ruby*, *.Net*, etc. A partir da versão 3.0 tornou-se *opensource*. Além disso, dispõem de uma IDE (*Flex Builder*, ferramenta paga) que auxilia o desenvolvimento de aplicações. As aplicações construídas utilizando o *Adobe Flex* são executadas no *FlashPlayer* (presente na maioria dos navegadores *web* modernos). Devido a isso, o *Adobe Flex* vem se tornando uma das linguagens mais utilizadas para desenvolvimento de RIAs, pois não apresenta incompatibilidade entre navegadores e pode ser executado em praticamente todos os navegadores disponíveis no mercado (ADOBE SYSTEMS, 2008[c].

A partir da análise realizada sobre as principais tecnologias disponíveis para desenvolvimento de RIAs, optou-se pelo *Adobe Flex* para desenvolver os componentes

[26] *Integrated Development Environment*. Conjunto de ferramentas que oferecem suporte ao desenvolvimento de software. Normalmente incluem editores de código, depuradores, etc.

[27] Pequeno aplicativo que adiciona funcionalidades extras a navegadores *web*.

gráficos do *framework*. A Tabela 16 apresenta a análise SWOT realizada sobre o *Adobe Flex*, que justifica a sua escolha para o desenvolvimento dos componentes gráficos do *framework*.

Tabela 16 – Análise SWOT do *Adobe Flex*

Forças	**Fraquezas**
1. Multiplataforma; 2. Pode ser executado em praticamente todos os navegadores *web* disponíveis no mercado; 3. É a tecnologia mais madura para o desenvolvimento de RIAs; 4. Possui uma IDE que auxilia o desenvolvimento de aplicações; 5. Pode trabalhar em conjunto com diversas linguagens de programação (*Java*, *Ruby*, *.Net*) 6. É a plataforma para desenvolvimento de RIAs que apresenta maior produtividades durante o processo de desenvolvimento 7. Facilidade no aprendizado da linguagem MXML e *ActionScript*. 8. Redução da curva de aprendizagem dos programadores.	1. Suporta somente a linguagem de desenvolvimento ActionScritp 3.0; 2. Depende do *flash player* instalado nas maquinas; 3. O desenvolvimento de interfaces muito complexas pode causar problemas de performance durante o download da aplicação pelo cliente.
Oportunidades	**Ameaças**
1. Esta sendo adotada como padrão de desenvolvimento por várias grandes empresas; 2. Tem uma boa aceitação pelas comunidades de desenvolvimento *web* na Internet; 3. Embora seja *opensource*, conta com o apoio da *Adobe* para manter e desenvolver a tecnologia.	1. Projeto pode ser descontinuado; 2. Aplicativos desenvolvidos utilizando o Flex 2 ou 3 podem ser incompatíveis com as versões futuras do *Adobe FlashPlayer*. 3. Falta de infra-estrutura de conexão com a Internet que suporte as aplicações.

Fonte: Kamyla Estuqui Parrado (2008)

A Tabela 17 descreve o cenário arquitetural de *log* de atividades. Um *log* (ou *logging*) nada mais é do que um conjunto de mensagens que registram o comportamento de uma

aplicação. O *log* de dados, ou atividades, normalmente é utilizado para depurar aplicações. O *log* de atividades permite aos administradores do sistema monitorar as ações realizadas pelos usuários do sistema. O objetivo de manter um *log* de atividades é registrar todas as interações entre os sistemas e os atores. Através dos arquivos de *log* é possível:

1. Identificar o motivo das falhas que ocorrem no sistema;
2. Auxiliar os usuários do sistema a entenderem o que está ocorrendo no sistema quando algum tipo de falha ocorre;
3. Auxiliar ao suporte técnico do sistema a identificar o motivo da ocorrência de falhas;
4. Registrar todas as alterações realizadas nos dados manipulados pelos sistemas;
5. Permitir que usuários e desenvolvedores de aplicações saibam quais foram os recursos e dados acessados no sistema.

Tabela 17 – Cenário arquitetural: *Log* de atividades

Identificador	Log de atividades
Cenário	Quando ocorrem falhas no sistema, os *logs* de atividades podem ser enviados por e-mail para os desenvolvedores do sistema para que estes possam efetuar as devidas correções. Esse processo é semelhando ao que ocorre no *Agines Network*: quando algum tipo de erro ocorre, um e-mail é enviado para os desenvolvedores informando o nome e o IP da máquina onde ocorreu o erro, o nome da tela e do método que causou o erro e uma breve descrição do erro. Com esses dados em mão, os desenvolvedores efetuam as correções necessárias antes mesmo que o cliente perceba os danos causados pelas falhas.
Problemas	**Perspectiva do usuário**
	1. Administradores de sistemas precisam de ferramentas que permitam monitorar as atividades realizadas pelos usuários do sistema. No entanto, quase sempre os sistemas não disponibilizam essas ferramentas. 2. As mensagens exibidas aos usuários, quanto algum tipo de falha ocorre, não são padronizadas. A mesma falha pode ocorrer em diferentes telas e exibir diferentes mensagens. 3. Nem sempre as mensagens exibidas são compreensíveis aos usuários, dessa forma, o usuário não sabe como proceder após uma falha.

	4. As mensagens não apresentam detalhes técnicos que possam auxiliar o suporte do sistema a identificar qual erro ocorreu e como solucionar o problema. 5. Ao enviar e-mail para os desenvolvedores, quando algum erro ocorre, o sistema pára de responder por alguns segundos, dando a impressão de que está travado.
	Perspectiva do desenvolvimento de software
	1. Manter um *log* de atividades pode influenciar o desempenho do sistema; 2. O *log* pode se tornar grande de mais e a consulta ao mesmo pode se tornar inviável; 3. As instruções necessárias para geração de *log* podem poluir o código da aplicação;
Solução proposta	Construir um mecanismo que registre as atividades do sistema e as interações entre sistemas e usuários. Além disso, esse mecanismo auxiliará o gerenciamento das falhas que ocorrem no sistema, permitindo aos usuários entenderem o que está ocorrendo e auxiliando o suporte técnico a identificar o motivo e a solução para as falhas.
Requisitos arquiteturais	▪ **Desempenho**: como as informações estarão sendo inseridas no arquivo de *log* enquanto o usuário utiliza o sistema, essa operação deve ocorrer de forma transparente, pois, a geração do arquivo de *log* não pode influenciar as interações entre sistema e usuários;
Arquiteturas candidatas	Construção de um mecanismo que registre todas as interações entre o sistema e os atores utilizando APIs disponíveis no mercado que são especializadas em gerar *log* de atividades.

Fonte: Kamyla Estuqui Parrado (2008)

Existem diversos *frameworks* e APIs desenvolvidos em *Java* para geração de *log*, no entanto, duas dessas APIs se destacam:

- ***Log4J***: é uma API *opensource* que faz parte do Projeto *Logging* mantido pela *Apache Foundation*. Foi desenvolvida para geração de *log* em aplicações *Java*, porém, se popularizou tanto que foi portada para outras linguagens como *Microsoft .Net*, *Python*, *C++*, etc. Normalmente o *Log4J* é utilizado para depurar aplicações, no entanto, o objetivo da API é registrar o comportamento de uma aplicação através de mensagens.

A API permite que as mensagens sejam salvas em arquivos, enviadas por e-mail, entre outras opções. O *Log4J* é rápido e não atrapalha o desempenho da aplicação. Um dos diferenciais dessa API é que ela pode ser configurada através de arquivos XML ou arquivos de propriedades [28]. Dessa forma, não é necessário poluir o código da aplicação com as instruções necessárias para geração de *log*. Além disso, a configuração através de arquivos permite que o *log* seja ativado/desativado em tempo de execução, sem que seja necessário alterar o código-fonte da aplicação (LOG4J, 2008).

- **API *Logging* do *Java***: é a API nativa para geração de *log* da plataforma *Java*. A API foi desenvolvida para impactar o mínimo possível no desempenho das aplicações. Devido a isso, a API define dinamicamente quais serão as mensagens exibidas ou não. O mecanismo utilizado pela API permite que outras APIs de *log* utilizadas pela aplicação trabalhem em conjunto com a esta API. O *Logging* disponibiliza interfaces e superclasses que podem ser estendidas para a criação de classes de *log* personalizadas (JAVA COMMUNITY PROCESS, 2008).

  A Tabela 18 apresenta a análise SWOT da alternativa escolhida: API *Log4J*.

Tabela 18 – Análise SWOT da API *Log4J*.

Forças	**Fraquezas**
1. Fácil de usar; 2. Não causa danos ao desempenho da aplicação; 3. Pode ser configurada através de arquivos de propriedades ou XML, diminuindo a quantidade de código que precisa ser inserido na aplicação para geração de *log*; 4. Permite ativar/desativar o *log* em tempo de execução, apenas alterando um arquivo de configuração.	1. Não identificadas.
Oportunidades	**Ameaças**
1. Ampla aceitação do mercado; 2. Utilização de uma das APIs mais flexíveis	1. Como todo projeto *opensoure*, este também pode ser descontinuado.

[28] Arquivos de propriedades, em *Java*, são arquivos com a extensão "*.properties*" onde o conteúdo é apresentado no formato chave=valor.

do mercado para geração de *log* em *Java*; 3. O projeto é mantido pela *Apache Foundation*, que é a fundação responsável por diversos outros projetos como o servidor *web Apache HTTP* e o servidor JEE *Apache Tomcat*.	

Fonte: Kamyla Estuqui Parrado (2008)

A Tabela 19 apresenta o cenário arquitetural de internacionalização de aplicações *web*. A internacionalização de uma aplicação envolve, também, a sua localização. De acordo com o W3C[29] (2008), a localização envolve a personalização de uma aplicação relacionada a:

1. Formatos numéricos, de data e de hora;
2. Uso de moeda;
3. Texto e gráficos contendo referências a objetos, ações ou idéias que, em determinada cultura, podem estar sujeitos à interpretação incorreta;
4. Exigências legais diversas, entre outros.

A internacionalização é definida pelo W3C (2008) como a criação de aplicações que permitem a fácil localização dos usuários da aplicação em termos de cultura, região ou idioma. Normalmente, internacionalizar uma aplicação envolve:

1. Criar e desenvolver aplicações de modo a eliminar os problemas para a localização ou implantação internacional.
2. Fornecer suporte aos recursos que não podem ser utilizados antes do término do processo da localização, como, por exemplo, suporte para texto vertical ou outros recursos tipográficos não latinos.
3. Oferecer suporte para formatos de data e hora, calendários locais, formatos de número e sistemas de numeração, etc.
4. Permitir que as alternativas localizadas sejam carregadas de acordo com as necessidades dos usuários da aplicação.

Normalmente a internacionalização é escrita como *i18n*, onde o *18* representa as dezoito letras existentes entre o *i* e *n* (*internationalization*, em inglês). Já a localização normalmente é representada por *l10n*, onde o *10* representa as dez letras existentes entre o *l* e o *n* (*localization*, em inglês).

[29] W3C (*World Wide Web Consortium*) é um consórcio internacional de empresas responsável pelo desenvolvimento e manutenção de diversos padrões relacionados à *web*, incluindo padrões para *web services*, internacionalização, XML, entre outros.

O W3C (2008) afirma que “a internacionalização ocorre como uma etapa fundamental no processo de criação e desenvolvimento, e não como uma consideração tardia que geralmente pode envolver uma re-engenharia complexa e dispendiosa”. O objetivo do cenário arquitetural de internacionalização é, através dos padrões propostos pelo W3C, identificar as possíveis soluções para o desenvolvimento de aplicações internacionalizadas.

Tabela 19 – Cenário arquitetural: Internacionalização

Identificador	Internacionalização
Cenário	Sistemas desenvolvidos para a *web* podem ser acessados em diversas partes do mundo por usuários que falam idiotas diferentes, que utilizam sistemas de medida, numéricos e monetários totalmente diferentes. Por isso, sistemas *web* devem fornecer mecanismos que permitam ao sistema se adaptar as normas de cada localidade onde está sendo acessado.
Problemas	**Perspectiva do usuário**
	1. Tradução inadequada; 2. Baixo desempenho durante a internacionalização;
	Perspectiva do desenvolvimento de software
	1. Processo complexo de tradução.
Solução proposta	Construção de um mecanismo de internacionalização que permita, através de um perfil de usuário, exibir o sistema com as configurações adequadas para a localidade onde está sendo acessado.
Requisitos arquiteturais	▪ **Desempenho**: a internacionalização do sistema ocorrerá logo após o *logon* do usuário no sistema e, esse processo não pode ser lento. Assim que o usuário for autenticado, o sistema deve ser exibido seguindo as configurações do usuário.
Arquiteturas candidatas	Utilização dos padrões propostos pelo W3C com os mecanismos de internacionalização fornecidos pelo *Adobe Flex* 3 ou pela plataforma *Java*.

Fonte: Kamyla Estuqui Parrado (2008)

Foram identificadas duas alternativas para implementar os padrões propostos pelo W3C:

- ***Adobe Flex 3***: até o *Flex* 2, os recursos necessários para localizar uma aplicação ficavam dentro do código-fonte da aplicação. Então, para que uma aplicação atendesse a diversas configurações de localização, era necessário compilar várias vezes a mesma aplicação, cada uma para uma localização diferente. Já com o *Flex* 3, o processo de localização se tornou mais flexível. Os recursos necessários para localizar a aplicação podem ser mantidos fora do código-fonte. Dessa forma, a aplicação passa a suportar múltiplas localizações. No entanto, uma desvantagem de utilizar os mecanismos de localização do *Adobe Flex* 3 é que, se houver a necessidade substituir a tecnologia utilizada pela interface gráfica, será necessário construir um novo mecanismos de internacionalização. Pois, os recursos disponibilizados pelo *Flex* 3 só podem ser utilizados no *Flex* 3, e isso dificulta a migração da interface gráfica do sistema para outras tecnologias (ADOBE SYSTEMS, 2008[a]).
- **Plataforma *Java***: disponibiliza mecanismos especializados em tratamento de data e hora, formatos numéricos e monetários. Permite conversão entre os tipos de caracteres utilizados de acordo com a localização. (SUN MICROSYSTEMS, 2008).

Assim como as técnicas de internacionalização presentes no *Flex* 3, as técnicas existentes no *Java* só podem ser utilizadas com a plataforma *Java*. Mas, ao contrário do *Flex* 3, o *Java* não irá limitar a aplicação, pois um dos requisitos para utilização do *framework* é que as aplicações sejam escritas em *Java*. Dessa forma, a tecnologia utilizada para desenvolver a interface gráfica das aplicações poderá ser substituída a qualquer momento sem a necessidade de reescrever todo o mecanismo de internacionalização. Porém, ao optar pela construção do mecanismo de internacionalização na camada *Java* do *framework*, pode ocorrer uma perda de desempenho durante o processo de internacionalização. Essa perda, no entanto, será compensada pela flexibilidade e portabilidade oferecidas pela plataforma *Java*.

A Tabela 20 apresenta a análise SWOT da alternativa escolhida para desenvolver o mecanismo de internacionalização das aplicações construídas a partir do *framework*.

Tabela 20 – Análise SWOT da internacionalização com *Java*.

Forças	**Fraquezas**
1. Flexibilidade; 2. Portabilidade.	1. Perda de desempenho; 2. Maior complexidade.
Oportunidades	**Ameaças**
1. Desenvolvimento de uma solução que atenda integralmente a todas as	1. Versões futuras da plataforma *Java* ou do SDK do *Flex* podem ser

necessidades da empresa.	incompatíveis entre si ou com as aplicações construídas com versões anteriores.

Fonte: Kamyla Estuqui Parrado (2008)

A Tabela 21 apresenta o cenário arquitetural para gerenciamento de relatórios. Todo sistema disponibiliza indicadores que são apresentados no formato de relatórios gerenciais que auxiliam a tomada de decisão. Hoje existem muitas ferramentas que auxiliam a construção de relatórios, tais como *iReport*, *CrystalReports*, etc.

Tabela 21 – Cenário arquitetural: Gerenciamento de relatórios

Identificador	Gerenciamento de relatórios
Cenário	Apesar das ferramentas gráficas que auxiliam a construção de relatórios, ainda é necessário fazer a integração entre a aplicação, o banco de dados e o relatório. Os usuários escolhem qual relatório desejam visualizar e fornecem os parâmetros necessários, que nada mais são do que os filtros do relatório. Esses parâmetros precisam ser enviados para o relatório para que este possa ser gerado corretamente.
Problemas	**Perspectiva do usuário**
	1. Relatórios muito lentos.
	Perspectiva do desenvolvimento de software
	1. Duplicação de código para efetuar chamada aos relatórios; 2. O sistema *Agines Network* apresenta às seguintes dificuldades: 2.1. Problemas para manter a grande quantidade de classes existentes para efetuar a chamada aos relatórios; 2.2. Constante aumento das classes necessárias para acessar os relatórios. Cada novo relatório inserido representa uma nova classe que precisa ser criada.
Solução proposta	Desenvolvimento de um mecanismo que receba os parâmetros necessários para gerar o relatório e repasse esses parâmetros ao relatório. Dessa forma, impedindo que cada relatório criado necessite de uma nova classe de controle.
Requisitos arquiteturais	▪ **Desempenho**: o tempo de resposta para exibição de um relatório não pode ser longo;

	▪ **Portabilidade**: os relatórios precisam ser portáveis a qualquer plataforma de software ou hardware, pois, por se tratar de um sistema *web*, não é possível prever o ambiente no qual o sistema será utilizado.
Arquiteturas candidatas	As principais opções disponíveis para construção de relatórios são: ▪ *Birt*; ▪ *JasperReport* / *iReport*; ▪ *CrystalReports*; ▪ Implementação proprietária utilizando as funcionalidades disponíveis na API *Graphics* do *Java*.

Fonte: Kamyla Estuqui Parrado (2008)

Existem muitas alternativas para o desenvolvimento dos relatórios de uma aplicação. Essas alternativas vão desde a utilização de ferramentas disponíveis no mercado até a construção de ferramentas proprietárias. As principais opções disponíveis são:

- ***Birt***: ferramenta voltada ao desenvolvimento de relatórios para aplicações web, em especial para a plataforma JEE. O *Birt* é composto por uma ferramenta gráfica (baseada na IDE *Eclipse*), para desenvolvimento visual dos relatórios, e um componente de tempo de execução que pode ser adicionado ao servidor de aplicações. Além disso, o *Birt* também oferece ferramentas para desenvolvimento de relatórios com gráficos. (BIRT, 2008).
- ***JasperReports* / *iReport***: *JasperReports* é uma API para definição e visualização de relatórios. A definição é feita utilizando arquivos XML que são compilados e dão origem a um arquivo com a extensão "*.jasper*". Esses arquivos, por sua vez, são interpretados pela API e podem dar origem a relatórios em diversos formatos (PDF, XML, HTML, CSV, XLS, RTF, TXT) ou apenas ser visualizado em tela, através de um visualizador próprio da API (JASPER FORGE[a], 2008). Embora o arquivo XML utilizado para gerar o arquivo "*.jasper*" possa ser editado manualmente, normalmente utiliza-se a ferramenta gráfica *iReport* para editar o XML. Através dessa ferramenta é possível desenhar relatórios complexos, utilizando tabelas, sub-relatórios, gráficos, e outros recursos disponibilizados pelo *JasperReports* (JASPER FORGE[b], 2008).
- ***CrystalReports***: é uma das ferramentas para construção de relatórios mais completas do mercado. No entanto, é uma ferramenta paga, que pode ser utilizada para visualizar

através de aplicações escritas em *Visual Basic* ou do próprio visualizador disponibilizado pela ferramenta (CRYSTAL REPORTS, 2008).

- **Implementação proprietária**: desenvolver uma solução proprietária para construção e visualização de relatórios implica em muito retrabalho, pois já existem diversas ferramentas que podem ser utilizadas para esta atividade. Além disso, muitos dos relatórios do *Agines Network* foram desenvolvidos utilizando uma implementação proprietária. Essa implementação é difícil de utilizar e, principalmente de manter. Não existe reutilização de código, pois cada relatório precisa ser codificado manualmente. Um exemplo disso é a impressão de Notas Fiscais do sistema. Para cada layout de nota é preciso criar uma nova classe que possui exatamente os mesmos métodos e atributos das demais classes, a única alteração é a posição dos campos que serão impressos. Outro problema é a inexistência de uma ferramenta gráfica que auxilie a construção de relatórios.

A Tabela 22 apresenta a análise SWOT das ferramentas de desenvolvimento *JasperReports* / *iReport*.

Tabela 22 – Análise SWOT das ferramentas *JasperReports* / *iReport*

Forças	**Fraquezas**
1. A API e a ferramenta gráfica são fáceis de utilizar; 2. A API permite exportar os relatórios para diversos formatos (HTML, PDF, RTF, etc); 3. O *iReport* dispõe de uma ampla gama de ferramentas para construir relatórios; 4. O *iReport* aumenta a produtividade ao permitir construir relatórios de forma gráfica.	1. A ferramenta gráfica *iReport*, antes da versão 3.0, era muito instável.
Oportunidades	**Ameaças**
1. Tanto a API *JasperReports* quanto a ferramenta gráfica *iReport* são muito utilizadas; 2. Existe muita documentação a respeito das funcionalidades presentes na API e na ferramenta gráfica.	1. Projeto pode ser descontinuado; 2. Normalmente as versões mais recentes da API *JasperReports* apresentam algum tipo de incompatibilidade com as versões anteriores.

Fonte: Kamyla Estuqui Parrado (2008)

A Tabela 23 descreve o cenário arquitetural de integração entre sistemas. Com o advento da Internet, as aplicações deixaram de ser executadas em computadores isolados e passaram a interagir com outras aplicações através da Internet. Isso deu origem aos sistemas distribuídos, ou seja, sistemas divididos em módulos que podem ser distribuídos entre servidores diferentes, mas que interagem entre si.

Além disso, hoje, no mercado existem diversos tipos de sistemas especializados nas mais diversas áreas. Muitas vezes uma empresa, ao adquirir um novo sistema, não deseja abandonar os sistemas já existentes. Dessa situação surge à necessidade de fazer sistemas distintos trabalharem em conjunto para atender aos requisitos dos clientes. O objetivo deste cenário arquitetural é encontrar alternativas viáveis para a construção de interfaces que permitam a sistemas externos trocar informações que as aplicações construídas a partir do *framework.*

Tabela 23 – Cenário arquitetural: Integração entre sistemas

Identificador	Integração entre sistemas
Cenário	O sistema *Agines Network* é integrado ao sistema *Simps PDV* para emissão de cupom fiscal. Essa integração ocorre através da funcionalidade "Aprovar Orçamento de Venda", disponível na tela "Orçamento de Vendas" do *Agines Network*. Ao aprovar um Orçamento de Venda gerando Cupom Fiscal, o *Agines Network* gera os dados necessários para emissão do Cupom no banco de dados do *Agines Network*. O sistema, então, faz uma cópia dos dados para o banco de dados do *Simps* PDV. Os pré-cupons gerados no *Agines Network* são emitidos através da funcionalidade "Emitir Cupom Fiscal", presente no *Simps* PDV. Quando isso ocorre, o *Simps* executa todas as operações necessárias para geração do cupom, de forma totalmente independente do *Agines Network*, ou seja, os processos realizados pelo *Simps* são totalmente desconhecidos pelo *Agines Network*. Para copiar as informações geradas no banco do *Simps* PDV para o banco do *Agines Network*, foi desenvolvido um pequeno aplicativo. Esse aplicativo é inicializado junto com o sistema operacional e, em períodos de tempos configurados no banco de dados do *Agines Network*, copia os dados presentes no banco do *Simps* PDV para o banco do *Agines Network*.

<table>
<tr><td rowspan="4">Problemas</td><td>Perspectiva do usuário</td></tr>
<tr><td>1. Necessidade de três programas diferentes para executar uma atividade.
2. Necessidade de deixar um programa executando o tempo todo para realiza a integração entre Simps PDV e Agines Network, pois, embora o aplicativo seja pequeno, ele consome memória e processamento.
3. Existe a possibilidade de o usuário fechar o aplicativo que leva os dados do Simps PDV para o Agines Network, dessa forma, as informações do Agines Network ficarão desatualizadas.
4. Curva de aprendizado maior, pois será necessário aprender a utilizar dois sistemas para realizar as tarefas necessárias.</td></tr>
<tr><td>Perspectiva do desenvolvimento de software</td></tr>
<tr><td>1. A má estruturação do sistema dificulta a integração entre as funcionalidades que já existem e as funcionalidades que precisam ser criadas;
2. O fato de o sistema ser construído utilizando applets inviabiliza a utilização de APIs e frameworks que possam auxiliar a integração entre os sistemas;</td></tr>
<tr><td>Solução proposta</td><td colspan="2">Construção de interfaces que permitam às aplicações construídas a partir do framework comunicar-se com sistemas externos de forma transparente, disponibilizando aos sistemas externos acesso aos dados e serviços disponíveis nas aplicações.</td></tr>
<tr><td>Requisitos arquiteturais</td><td colspan="2">▪ Desempenho: a comunicação entre sistemas deve ocorrer de forma totalmente transparente aos usuários dos sistemas. Para que isso ocorra, essa comunicação não pode influenciar no desempenho dos sistemas. Ou seja, os sistemas devem apresentar tempos de respostas iguais, tanto quando estiverem trabalhando sozinhos, quanto quando estiverem trabalhando em conjunto.
▪ Segurança: a comunicação entre dois sistemas tem que ocorrer de forma segura. Apenas sistemas autorizados podem se comunicar uns com os outros. Devem ser feitas validações para garantir que apenas os sistemas externos que possuam autorização consigam</td></tr>
</table>

	acessar os serviços disponíveis. ▪ **Portabilidade**: a solução desenvolvida para integrar sistemas precisa ser portável a qualquer plataforma de software ou hardware, pois, os sistemas com os quais a aplicação irá interagir podem estar em um ambiente totalmente diferente do ambiente da aplicação; ▪ **Tolerância a falhas**: sistemas *web* são acessados por diversos clientes ao mesmo tempo, dessa forma, as falhas causadas no sistema por um determinado cliente, não podem afetar a execução do sistema em outro cliente; ▪ **Interoperabilidade**: este é o princípio fundamental para a integração entre sistemas, ou seja, a interoperabilidade é a responsável por permitir que sistemas distintos atuem em conjunto de forma transparente ao usuário.
Arquiteturas candidatas	▪ Utilização de uma camada de *web services* para acesso aos serviços fornecidos; ▪ Utilização de outros mecanismos de integração de aplicações distribuídas, como CORBA, RMI, etc.

Fonte: Kamyla Estuqui Parrado (2008)

Para realizar a integração entre sistemas foram identificadas as seguintes alternativas:

- **CORBA**: esse já foi um dos principais representantes dos padrões utilizados para desenvolver aplicações distribuídas. No entanto, com o advento da *Service Oriented Architecture* (SOA), CORBA tem perdido bastante espaço. Isso se deve principalmente a simplicidade de SOA quando comparado a CORBA.
- **RMI**: embora o RMI seja uma forma nativa de trabalhar com objetos distribuídos em *Java*, ele apresenta a desvantagem de se limitar a *Java*. Ou seja, apenas aplicações que são executadas em uma máquina virtual *Java* podem utilizar RMI.
- ***Web services WS-**** : em *Java*, existem diversas opções para o desenvolvimento desse tipo de *web service*. As duas principais são: servidores de aplicação *Java* EE completos e *frameworks* independentes (como *Apache Axis*). O *Apache Axis* facilita muito o desenvolvimento de *web services*. O *Apache Axis* realiza o mapeamento de *schemas* XML para *Java* e vice-versa. O *Apache Axis* também é compatível com WSDL e possui abstrações do protocolo SOAP. Além disso, o *Apache Axis* não

necessita de um servidor *Java* EE completo para funcionar, ele pode ser utilizado em conjunto com servidores leves como *Apache Tomcat* ou *Jetty* (PEREIRA, 2008).

- ***Web services RESTful***: o desenvolvimento deste tipo de *web service*, em *Java*, conta com a JSR[30]-311 e o *Jersey*. A JSR-311 (*JAX-RS: Java API for RESTful Web Services*) é uma especificação para Java EE (versão 6), cujo objetivo é fornecer suporte ao desenvolvimento de *web services RESTful* na plataforma *Java*. De acordo com Buback e Pereira (2008), o objetivo principal da JSR-311 é trazer uma API fácil de utilizar e que simplifique o desenvolvimento sem reduzir o poder que este tipo de *web service* oferece. A JSR-311 permite trabalhar com POJOs e fornece uma série de anotações para facilitar o processo de desenvolvimento. O projeto *Jersey* é uma implementação das especificações da JSR-311.

A Tabela 24 apresenta a análise SWOT da alternativa escolhida para o desenvolvimento da integração entre sistemas: *web services RESTful*.

Tabela 24 – Análise SWOT dos *web services RESTful*

Forças	**Fraquezas**
1. Simplicidade: se comparado aos *web services* WS-*, os *web services RESTful* são muito mais simples de desenvolver; 2. Utiliza duas tecnologias que já estão no mercado há muito tempo e que são comprovadamente eficientes: HTTP e XML; 3. Por ser bem mais simples do que os *web services* WS-*, existem poucos padrões que precisam ser seguidos para o desenvolvimento de *web services RESTful*; 4. Interoperabilidade entre os padrões de *web services*; 5. Permite o desenvolvimento de aplicações consumidoras em qualquer plataforma.	1. Existem poucas IDEs que oferecem suporte ao desenvolvimento de *web services RESTful*.
Oportunidades	**Ameaças**
1. Utilização de uma tecnologia que está	1. Nem todas as linguagens de programação

[30] *Java Specification Request*.

simplificando a implementação de SOA através de *web services*.	oferecem suporte direto ao desenvolvimento de *web services RESTful*. Ou seja, nem todas as linguagens dispõem de ferramentas e APIs que facilitem o desenvolvimento deste tipo de *web service*.

Fonte: Kamyla Estuqui Parrado (2008)

A Tabela 25 apresenta o cenário arquitetural de fluxo de controle entre *framework* e aplicações. O objetivo deste cenário é identificar soluções para que o *framework* assuma o fluxo de controle da aplicação e devolva o fluxo quando for necessário.

Tabela 25 – Cenário arquitetural: Fluxo de controle entre *framework* e aplicações

Identificador	Fluxo de controle entre framework e aplicações
Cenário	Existem certos tipos de *frameworks* que seguem o princípio de *Hollywood*, também conhecido como inversão de controle (IoC – *Inversion of Control*): "*Don't call us. We call you.*" (Não nos chame. Nós chamamos você). Ou seja, o *framework* assume o controle da aplicação, dessa forma, determinando o fluxo de controle da aplicação. Normalmente isso ocorre quando alguma classe da aplicação implementa uma interface ou estende uma classe abstrata do *framework*. Essa classe da aplicação estende alguma classe abstrata do *framework* que contém um algoritmo parcialmente implementado (classes *Template*). Esse algoritmo indica ao *framework* em que momentos ele deve assumir o controle e, em quais momentos a aplicação pode, ou deve, sobrescrever ou implementar os métodos desse algoritmo (métodos *Hooks*).
Problemas	**Perspectiva do usuário**
	Os desenvolvedores de aplicações, ou seja, usuários do *framework* são os maiores afetados pelos problemas deste cenário. O maior deles é a complexidade que os algoritmos podem adquirir. O objetivo de um *framework* é se manter o mais simples possível. No entanto, devido à necessidade de disponibilizar uma série de funcionalidades pré-determinadas e, ainda ser flexível para suportar funcionalidades não

	previstas no escopo original, o algoritmo que determina o fluxo de controle entre aplicação e *framework* pode se tornar bastante complexo.
	Perspectiva do desenvolvimento de software
	1. Dificuldade para determinar em quais pontos o *framework* assumirá o controle da aplicação e em quais pontos a aplicação terá autonomia para se controlar sozinha. 2. Dificuldade para desenvolver algoritmos flexíveis, ou seja, algoritmos que dirijam a arquitetura da aplicação, mas que possam ser modificados para comportar as necessidades específicas de cada aplicação.
Solução proposta	Construção de algoritmos que determinem o fluxo de controle entre a aplicação e o *framework* e que sejam flexíveis o suficiente para atender aos requisitos dos sistemas construídos a partir do *framework*.
Requisitos arquiteturais	▪ **Flexibilidade**: em um cenário como este, a flexibilidade se faz necessária para que as classes *Template* e os métodos *Hook* atendam de forma eficaz os requisitos das aplicações desenvolvidas a partir do *framework*; ▪ **Usabilidade**: pelo fato de determinar boa parte da estrutura das aplicações, a usabilidade é um requisito de extrema importância para este cenário, pois, se este cenário não for inteiramente compreendido pelos usuários do *framework*, existem grandes chances dos usuários do *framework* não utilizarem este cenário, ou tentarem burlar o que foi estabelecido nas classes *Template*;
Arquiteturas candidatas	Desenvolvimento proprietário de mecanismos de inversão de controle.

Fonte: Kamyla Estuqui Parrado (2008)

A Tabela 26 apresenta a análise SWOT para o desenvolvimento proprietário de mecanismos de inversão de controle.

Tabela 26 – Análise SWOT do desenvolvimento proprietário de *IoC*

Forças	**Fraquezas**
1. Domínio total sobre o código-fonte e as	1. A solução pode se tornar muito complexa.

funcionalidades.	
Oportunidades	**Ameaças**
1. Construir uma solução que pode ser reutilizada em projetos futuros.	1. A mudança de requisitos pode afetar negativamente o fluxo de controle do *framework*.

Fonte: Kamyla Estuqui Parrado (2008)

3.3.2. Atores e casos de uso

Com base nos cenários arquiteturais desenvolvidos, foram identificados os atores e casos de uso do *framework*. Os atores são as entidades externas que interagem de alguma forma com o *framework*. Os casos de uso são as funcionalidades disponíveis no *framework*. A Tabela 27 descreve os atores envolvidos com o *framework*.

Tabela 27 – Atores do *framework*

Definição dos atores		
ID	**Ator**	**Função**
1	Aplicação	São os sistemas construídos a partir do *framework*.
2	Usuário da aplicação	São os usuários que utilizam os sistemas construídos a partir do *framework*.
3	Sistema externo	São sistemas construídos, ou não, a partir do *framework*, mas que interagem com sistemas construídos a partir do *framework*.

Fonte: Kamyla Estuqui Parrado (2008)

Alguns cenários arquiteturais deram origem a mais de um caso de uso. Como é o caso do cenário *Log* de atividades, esse cenário deu origem aos casos de uso Manter *log* de atividades, Enviar e-mail e Gerenciar falhas. A Tabela 28 descreve o objetivo dos casos de uso identificados.

Tabela 28 – Casos de uso do *framework*

Definição dos casos de uso		
ID	**Caso de uso**	**Objetivo**
1	Gerenciar fluxo de controle	Determinar o fluxo de controle entre a aplicação e o *framework*. Ou seja, determina em quais momentos o *framework* assume o fluxo de controle da aplicação e em

		quais momentos a aplicação tem autonomia para determinar o fluxo de controle.
2	Persistir dados	Tratar a criação e manipulação das informações armazenadas em bancos de dados, bem como realizar o mapeamento objeto-relacional.
3	Integrar sistemas	Disponibilizar interfaces para que sistemas externos interajam com as aplicações construídas a partir do *framework*.
4	Autenticar usuários	Permitir que apenas usuários com autorização tenham acesso aos serviços e informações disponibilizados pelas aplicações construídas a partir do *framework*.
5	Gerenciar falhas	Tratar as possíveis falhas que podem ocorrer durante a execução das aplicações construídas a partir do *framework*.
6	Internacionalizar aplicações	Permitir às aplicações serem visualizadas utilizando configurações de localização diferentes, de acordo com o perfil do usuário.
7	Manter *log* de atividades	Registrar as interações entre as aplicações e seus usuários.
8	Manter perfil de usuários	Permitir aos usuários da aplicação personalizar características do sistema, tais como página inicial, idioma padrão, etc.
9	Enviar e-mail	Permitir à aplicação, ou ao usuário da aplicação, enviar e-mails.
10	Gerenciar relatórios	Gerenciar a visualização dos relatórios disponibilizados pelas aplicações construídas a partir do *framework*.

Fonte: Kamyla Estuqui Parrado (2008)

3.3.3. Priorização dos casos de uso

Ao iniciar o desenvolvimento de um sistema, não é possível implementar todas as funcionalidades de uma única vez. Esse processo deve ocorre iterativamente, iniciando com os casos de uso mais importantes e chegando até os casos de uso de prioridade baixa.

A priorização dos casos de uso visa determinar quais são os casos de uso mais críticos e que devem ser implementados primeiro. No entanto, é preciso levar em consideração que os

requisitos dos sistemas estão em constante mudança. Portanto, a priorização dos casos de uso não é um processo que ocorre apenas uma vez. Conforme os requisitos dos sistemas mudam, a priorização pode mudar também. Inicialmente os casos de uso do *framework* são priorizados da seguinte forma:

1. Persistir dados;
2. Manter *log* de atividades;
3. Autenticar usuários;
4. Gerenciar falhas;
5. Internacionalizar aplicações;
6. Manter perfil de usuários;
7. Enviar e-mail;
8. Gerenciar relatórios;
9. Gerenciar fluxo de controle;
10. Integrar sistemas.

Conforme o *framework* for utilizado pelas aplicações construídas a partir dele, novos casos de uso podem surgir e serão incluídos na priorização dos casos de uso.

3.3.4. Especificação dos casos de uso

Após a identificação e priorização dos casos de uso, estes precisam ser detalhados. Nesta seção os casos de uso identificados são especificados, determinando quais são as regras de negócios envolvidas com cada caso de uso, quais atores interagem com qual caso de uso, etc.

A Tabela 29 demonstra o modelo utilizado para fazer o detalhamento dos casos de uso. As demais tabelas desta seção detalham os casos de uso identificados.

Tabela 29 – Modelo de especificação de caso de uso

Caso de uso	Define um nome para caso de uso. Deve ser expresso em poucas palavras e estar diretamente associado ao objetivo do caso de uso.
Objetivo	Descreve brevemente como o caso de uso funciona. Deve deixar claro qual o objetivo do caso de uso e permitir seu entendimento sem a necessidade de ler toda a especificação do caso de uso.
Importância	Determina o grau de importância que o caso de uso tem. Permite priorizar quais são os casos de uso críticos do sistema, ou seja,

	aqueles que precisam ser implementados primeiro.
Ator principal	É o responsável pela inicialização do caso de uso.
Ator(es) secundário(s)	Interage com o caso de uso através do envio ou recebimento de informações.
Ativação	Indica qual ação dará inicio ao caso de uso.
Pré-condições	Descreve os requisitos necessários para que o caso de uso possa ser iniciado.
Fluxo principal	
Representa o cenário principal de interação entre o sistema e os atores. Não existem normas para a descrição dos fluxos dos casos de uso. No entanto existem alguns padrões propostos. Será utilizado para descrever o fluxo principal a descrição numerada onde cada passo do ator ou do sistema é descrito em uma linha que recebe uma numeração seqüencial.	
Fluxo(s) alternativo(s)	
Representam os caminhos alternativos que um caso de uso pode tomar ao ser executado.	
Fluxo(s) de exceção	
Representam as exceções das regras dos casos de uso e as ações que devem ser tomadas caso essas regras não sejam respeitadas.	
Pós-condições	Descreve o estado que o sistema deve estar após a execução do caso de uso.
Regras	Descrevem as restrições funcionais que o caso de uso deve respeitar durante a sua execução. Podem se referir ao domínio do negócio, domínio da aplicação ou domínio do processo.
Pontos de extensão	Representam os casos de uso que estendem o caso de uso base e em que ponto eles são chamados dentro dos fluxos de evento.

Fonte: Kamyla Estuqui Parrado (2008)

A Tabela 30 apresenta a especificação do caso de uso Gerenciar fluxo de controle. Esse caso de uso originou-se do cenário arquitetural Gerenciamento de fluxo de controle.

Tabela 30 – Especificação do caso de uso Gerenciar fluxo de controle

Caso de uso	Gerenciar fluxo de controle
Objetivo	Determinar o fluxo de controle entre a aplicação e o *framework*. Ou seja, determina em quais momentos o *framework* assume o fluxo de controle da aplicação e em quais momentos a aplicação tem autonomia para determinar o fluxo de controle.

Importância	Risco alto, prioridade baixa.
Ator principal	Aplicação.
Ator(es) secundário(s)	Não se aplica.
Ativação	Inicialização da aplicação.
Pré-condições	As classes de controle da aplicação devem estender alguma classe abstrata do *framework*.

Fluxo principal

1. Após ser inicializada, a aplicação transfere o fluxo de controle para o *framework*. O *framework* então:
   1.1. Autentica o usuário da aplicação;
   1.2. Carrega e aplica o perfil do usuário na aplicação;
   1.3. Internacionaliza a aplicação.
2. Usuário navega pelo sistema. Para cada tela acessada pelo usuário:
   2.1. *Framework* executa o que está previsto no comportamento padrão da classe abstrata que foi estendida;
   2.2. Aplicação sobrescreve os métodos *hook* da classe abstrata estendida.

Fluxo(s) alternativo(s)

Não se aplica.

Fluxo(s) de exceção

Não se aplica.

Pós-condições	Os métodos *hooks* de uma classe abstrata do *framework* foram sobrescritos.
Regras	Não se aplica.
Pontos de extensão	1. **Gerenciar falhas**: se algum tipo de falha ocorrer, então, o fluxo de controle será encaminhado para o caso de uso Gerenciar falhas. 2. **Manter *log* de atividades**: todas as ações realizadas pelo sistema são incluídas em um *log* de atividades. 3. **Internacionalizar aplicações**: após ser inicializada e o usuário da aplicação ter sido autenticado, a aplicação deve ser internacionalizada. 4. **Manter perfil de usuário**: após o usuário da aplicação ser autenticado, o perfil do usuário deve ser aplicado ao sistema.

Fonte: Kamyla Estuqui Parrado (2008)

A Tabela 31 apresenta a especificação do caso de uso Persistir dados, originado do cenário arquitetural Manipulação de dados.

Tabela 31 – Especificação do caso de uso Persistir dados

Caso de uso	Persistir dados
Objetivo	Tratar a criação e manipulação das informações armazenadas em bancos de dados, bem como realizar o mapeamento objeto-relacional.
Importância	Risco alto, prioridade alta.
Ator principal	Usuário da aplicação.
Ator(es) secundário(s)	Aplicação.
Ativação	Usuário da aplicação efetua *logon* no sistema.
Pré-condições	1. O banco de dados da aplicação precisa existir.
Fluxo principal	

1. Usuário da aplicação preenche uma interface de entrada de dados e pressiona o botão "Salvar";
2. Aplicação verifica se os dados preenchidos estão corretos:
   2.1. Se os dados estiverem corretos, então aplicação cria um objeto com os dados fornecidos.
   2.2. Senão, se os dados estiverem incorretos ou algum campo obrigatório não estiver preenchido, então aplicação mostrará uma mensagem ao usuário da aplicação informando o que aconteceu;
3. Aplicação cria um objeto do tipo adequado e envia para o *framework*;
4. *Framework* verifica se as tabelas do banco existem:
   4.1. Se as tabelas não existirem, então *framework* cria as tabelas.
   4.2. Senão, se as tabelas existirem, então *framework* verifica qual operação realizar:
      4.2.1. Se a operação for de inclusão, então *framework* cria um objeto no banco;
      4.2.2. Senão, se a operação for de atualização, então *framework* atualiza o objeto no banco;
      4.2.3. Senão, se a operação for de exclusão, então *framework* verifica se o objeto está sendo utilizado em alguma outra parte do sistema. Se não estiver, *framework* exclui o objeto. Senão, se o objeto estiver sendo utilizado, então o *framework* retorna um valor indicando que não é possível excluir o objeto em questão (ou FA01);

4.2.4. Senão, se a operação for de consulta, *framework* procura pelo objeto que atenda aos parâmetros enviados. Se o *framework* não encontrar nenhum objeto correspondente aos parâmetros, então retorna um valor indicando que o objeto não foi encontrado. Senão *framework* retorna o(s) objeto(s) encontrado(s).

4.2.5. Se a operação for de inclusão, atualização ou exclusão e todo o processo ocorrer corretamente, então aplicação exibe uma mensagem informando que a operação foi realizada com sucesso. Senão, se a operação for de consulta, então aplicação apresenta o(s) objeto(s) encontrado(s) na interface gráfica do sistema.

5. Se alguma falha ocorrer, então fluxo de controle é encaminhado para o caso de uso Gerenciar falhas.

Fluxo(s) alternativo(s)
FA01 - Existem situações onde os dados não deverão ser excluídos, e sim inativados. Se o *framework* identificar, através de um parâmetro, que os dados não devem ser excluídos, então, *framework* deve inativar o objeto atribuindo o valor "*false*" ao atributo "ativo" do objeto em questão.

Fluxo(s) de exceção
Não se aplica.

Pós-condições	1. Um objeto foi criado ou atualizado, ou excluído do banco. Ou um objeto, ou conjunto de objetos, foi trazido do banco de dados para a interface gráfica.
Regras	1. Todos os dados devem ser validados antes da aplicação criar o objeto que será enviado para o *framework*. A validação dos dados consiste em: 1.1. Verificar se todos os dados necessários para criar um determinado objeto foram preenchidos pelo usuário da aplicação; 1.2. Verificar se o tipo de dado informado corresponde ao tipo de dado esperado por um determinado atributo do objeto em questão. 2. Ao excluir um objeto, deve ser verificado se este objeto não está sendo utilizado em nenhum outro lugar do sistema. Se o objeto estiver sendo utilizado, não permitir a exclusão.

	3. Ao excluir um objeto, o *framework* deve verificar se este pode ser excluído, ou se o objeto deve apenas ser inativado.
Pontos de extensão	1. **Gerenciar falhas**: se algum tipo de falha ocorrer, então, o fluxo de controle será encaminhado para o caso de uso Gerenciar falhas. 2. **Manter *log* de atividades**: todas as ações realizadas pelo sistema são incluídas em um *log* de atividades.

Fonte: Kamyla Estuqui Parrado (2008)

A Tabela 31 apresenta a especificação do caso de uso Integrar sistemas, originado do cenário arquitetural Integrar sistemas.

Tabela 32 – Especificação do caso de uso Integrar sistemas

Caso de uso	Integrar sistemas
Objetivo	Disponibilizar interfaces para que sistemas externos interajam com as aplicações construídas a partir do *framework*.
Importância	Risco alto, prioridade baixa.
Ator principal	Sistema externo.
Ator(es) secundário(s)	Aplicação.
Ativação	Sistema externo envia uma mensagem à aplicação solicitando acesso a algum serviço.
Pré-condições	O sistema externo deve ter autorização para interagir com a aplicação.
Fluxo principal	
1. Sistema externo envia uma solicitação de autorização para acessar os serviços da aplicação; 2. Enquanto o sistema externo estiver se comunicando com a aplicação, *framework* registra as atividades no *log* de atividades do sistema; 3. *Framework* verifica se a solicitação é válida e se o sistema externo tem autorização para acessar serviços da aplicação: 3.1. Se a solicitação for válida então, *framework* verifica para quais serviços o sistema externo tem permissão de acesso e libera o acesso à esses serviços; 3.2. Senão, se a solicitação for inválida, então *framework* nega o acesso aos serviços e	

envia um *e-mail* aos desenvolvedores do sistema informando sobre a tentativa não autorizada de acesso; 4. Aplicação realiza o processamento necessário para executar o serviço solicitado; 5. *Framework* envia o resultado do processamento ao sistema externo; 6. *Framework* finaliza conexão entre os dois sistemas.
Fluxo(s) alternativo(s)
Não se aplica.
Fluxo(s) de exceção
Não se aplica.

Pós-condições	Uma interação entre um sistema externo e a aplicação ocorreu.
Regras	1. Todo sistema externo que tentar se comunicar com a aplicação precisa ter um código de segurança que deverá ser validado antes de iniciar a integração. A integração só ocorrerá se o resultado dessa validação for verdadeiro. 2. Toda a comunicação entre o sistema externo e a aplicação deve ser registrada em um *log* de atividades.
Pontos de extensão	1. **Gerenciar falhas**: se algum tipo de falha ocorrer, então, o fluxo de controle será encaminhado para o caso de uso Gerenciar falhas. 2. **Manter *log* de atividades**: todas as ações realizadas pelo sistema são incluídas em um *log* de atividades.

Fonte: Kamyla Estuqui Parrado (2008)

O cenário arquitetural *Log* de atividades deu origem a três casos de uso. O primeiro, Gerenciar falhas, é apresentado na Tabela 33.

Tabela 33 – Especificação do caso de uso Gerenciar falhas

Caso de uso	Gerenciar falhas
Objetivo	Tratar as possíveis falhas que podem ocorrer durante a execução das aplicações construídas a partir do *framework*.
Importância	Risco baixo, prioridade alta.
Ator principal	Aplicação.
Ator(es) secundário(s)	Usuário da aplicação.
Ativação	Algum tipo de falha ocorre na aplicação.

Pré-condições	Ocorrência de algum tipo de falha.
Fluxo principal	
1. Aplicação direciona o fluxo de controle para o *framework*; 2. *Framework* registra a falha em um *log* de atividades (através do caso de uso Manter log de atividades); 3. *Framework* envia o *log* de atividades por e-mail para os desenvolvedores da aplicação; 4. *Framework* verifica qual tipo de erro ocorreu; 5. *Framework* devolve o fluxo de controle à aplicação; 6. Aplicação exibe uma mensagem informando a falha ao usuário da aplicação.	
Fluxo(s) alternativo(s)	
Não se aplica.	
Fluxo(s) de exceção	
Não se aplica.	
Pós-condições	1. Uma mensagem foi exibida ao usuário informando o que ocorreu; 2. A descrição do que ocorreu foi inserida no arquivo de *log* do sistema; 3. Um e-mail foi enviado aos desenvolvedores da aplicação.
Regras	Os registros de *log* gerados pelo sistema não podem ser apagados.
Pontos de extensão	1. **Gerenciar falhas**: se algum tipo de falha ocorrer, então, o fluxo de controle será encaminhado para o caso de uso Gerenciar falhas. 2. **Manter *log* de atividades**: todas as ações realizadas pelo sistema são incluídas em um *log* de atividades.

Fonte: Kamyla Estuqui Parrado (2008)

A Tabela 34 apresenta a especificação do caso de uso Internacionalizar aplicações, originado do cenário arquitetural Internacionalização.

Tabela 34 – Especificação do caso de uso Internacionalizar aplicações

Caso de uso	Internacionalizar aplicações
Objetivo	Permitir às aplicações serem visualizadas utilizando configurações de localização diferentes, de acordo com o perfil

	do usuário.
Importância	Risco baixo, prioridade alta.
Ator principal	Aplicação.
Ator(es) secundário(s)	Usuário da aplicação.
Ativação	Autenticação do usuário.
Pré-condições	1. O usuário da aplicação deve logar no sistema através do caso de uso Autenticar usuário.

Fluxo principal

1. *Framework* procura pelo perfil do usuário ativo:
   1.1. Se o *framework* encontrar o perfil, então verifica qual o idioma padrão do usuário.
   1.2. Senão, se o usuário não possuir perfil ou o perfil não contiver um idioma selecionado, então a aplicação é exibida com as configurações de idioma padrão.
2. *Framework* procura pelo arquivo de internacionalização do idioma:
   2.1. Se o *framework* encontrar o arquivo, então altera o idioma e demais configurações da aplicação.
   2.2. Senão, se o arquivo de configuração referente ao idioma não for encontrado, o fluxo de controle deve ser direcionado para o caso de uso Gerenciar falhas.
3. Aplicação é exibida com as configurações adequadas.

Fluxo(s) alternativo(s)

Não se aplica.

Fluxo(s) de exceção

Não se aplica.

Pós-condições	Aplicação foi exibida com as configurações de internacionalização correspondentes ao idioma selecionado.
Regras	1. Toda aplicação deve possuir pelo menos um idioma padrão e um arquivo de configuração referente a este idioma; 2. Cada idioma pode ter somente um arquivo de configuração; 3. Cada usuário da aplicação pode ter apenas um idioma selecionado.
Pontos de extensão	1. **Gerenciar falhas**: se algum tipo de falha ocorrer, então, o fluxo de controle será encaminhado para o caso de uso Gerenciar falhas. 2. **Manter *log* de atividades**: todas as ações realizadas pelo

	sistema são incluídas em um *log* de atividades. 3. **Autenticar usuário**: o usuário precisa estar *logado* no sistema para que as configurações de autenticação sejam aplicadas.

Fonte: Kamyla Estuqui Parrado (2008)

A Tabela 35 apresenta o segundo caso de uso originado do cenário arquitetural *Log* de atividades: Manter *log* de atividades.

Tabela 35 – Especificação do caso de uso Manter *log* de atividades

Caso de uso	Manter log de atividades
Objetivo	Registrar as interações entre as aplicações e seus usuários.
Importância	Risco baixo, prioridade baixo.
Ator principal	Aplicação.
Ator(es) secundário(s)	Não se aplica.
Ativação	Inicialização da aplicação.
Pré-condições	Aplicação deve ser inicializada.
Fluxo principal	

1. *Framework* cria um arquivo de *log*;
2. *Framework* registra data e hora na inicialização do sistema no arquivo;
3. Usuário da aplicação efetua *login*;
4. *Framework* registra o nome do usuário que fez *logon* no sistema no arquivo, bem como a data e a hora do *logon*;
5. Para cada tela que o usuário acessar:
   5.1. *Framework* registra a data, a hora e o nome da tela que o usuário acessou;
   5.2. *Framework* registra cada operação realizada na tela:
      5.2.1. Se a operação for de consulta, então *framework* registra os parâmetros utilizados pela consulta;
      5.2.2. Se a operação for de edição, então *framework* registra o conteúdo anterior a edição e o novo conteúdo do registro;
      5.2.3. Se a operação for de inclusão ou exclusão, então *framework* registra a data, hora e nome do usuário que incluiu ou excluiu o registro;
6. Usuário efetua *logoff* ou fecha o sistema;
7. *Framework* registra data e hora do *logoff*;

8. *Framework* finaliza o arquivo de log. 9. *Framework* grava o arquivo no banco de dados; 10. *Framework* apaga o arquivo gerado.
Fluxo(s) alternativo(s)
Não se aplica.
Fluxo(s) de exceção
Não se aplica.

Pós-condições	Foi gerado um registro contendo as informações decorrentes da interação entre aplicação e usuário.
Regras	1. O nome do arquivo de *log* deve ser a data e a hora na qual o arquivo foi gerado no formato YYYYMMDD-HH.MM.SS. Isso é necessário para que o *framework* não gere dois arquivos com o mesmo nome e, conseqüentemente, sobrescreva um dos arquivos. 2. O *framework* não deve sobrescrever as informações em um arquivo de log, e sim, criar um novo sempre que for necessário. 3. As operações realizadas para manter o *log* devem ser realizadas em segundo plano, não interferindo na utilização do sistema.
Pontos de extensão	1. **Gerenciar falhas**: se algum tipo de falha ocorrer, então, o fluxo de controle será encaminhado para o caso de uso Gerenciar falhas.

Fonte: Kamyla Estuqui Parrado (2008)

A Tabela 36 apresenta a especificação do caso de uso Autenticar usuário, originado do cenário arquitetural Autenticar usuário.

Tabela 36 – Especificação do caso de uso Autenticar usuário

Caso de uso	Autenticar usuário
Objetivo	Permitir que apenas usuários com autorização tenham acesso os serviços e informações disponibilizados pelas aplicações construídas a partir do *framework*.
Importância	Risco alto, prioridade alta.

Ator principal	Usuário da aplicação.
Ator(es) secundário(s)	Aplicação.
Ativação	Inicialização da aplicação.
Pré-condições	Usuário da aplicação deve possuir *login* e senha validos.

Fluxo principal

1. Usuário da aplicação informa *login* e senha e pressiona o botão *logon*;
2. Aplicação envia os dados para o *framework*;
3. *Framework* efetua uma consulta ao banco de dados para procurar um usuário com o *login* informado. Se o usuário não for encontrado, então deverá ser exibida uma mensagem ao usuário informando que o *login* informado não foi encontrado;
4. Se *framework* encontrar o usuário então verifica se a senha informada é igual à senha gravada no banco.
5. Se as senhas forem iguais, então *framework* retorna um valor informando que a aplicação pode ser liberada. Senão, se as senhas não forem iguais, então deverá ser exibida uma mensagem ao usuário informando que a senha está incorreta.
6. Aplicação libera o acesso ao sistema.

Fluxo(s) alternativo(s)

Não se aplica.

Fluxo(s) de exceção

Não se aplica.

Pós-condições	O acesso ao sistema foi liberado ao usuário.
Regras	1. Se o usuário tentar logar no sistema e o usuário não for encontrado ou a senha estiver incorreta por mais de 5 vezes, então o sistema deverá ser bloqueado para este usuário; 2. As senhas devem conter no mínimo 6 caracteres alfanuméricos; 3. Todas as senhas devem ser criptografadas antes de serem gravadas no banco; 4. Nenhuma senha pode ser decodificada; 5. Para comparar se duas senhas são iguais, a senha que será comparada com a que está no banco deve ser criptografada;
Pontos de extensão	1. **Gerenciar falhas**: se algum tipo de falha ocorrer, então, o fluxo de controle será encaminhado para o caso de uso

	Gerenciar falhas. 2. **Manter *log* de atividades**: todas as ações realizadas pelo sistema são incluídas em um *log* de atividades.

Fonte: Kamyla Estuqui Parrado (2008)

A Tabela 37 apresenta a especificação do caso de uso Manter perfil de usuário.

Tabela 37 – Especificação do caso de uso Manter perfil de usuário

Caso de uso	Manter perfil de usuário
Objetivo	Permitir aos usuários da aplicação personalizar características do sistema, tais como página inicial, idioma padrão, etc.
Importância	Risco baixo, prioridade baixo.
Ator principal	Usuário da aplicação.
Ator(es) secundário(s)	Aplicação.
Ativação	Autenticação do usuário.
Pré-condições	Usuário deve estar *logado* no sistema.
Fluxo principal	
1. Usuário efetua *logon* no sistema; 2. Aplicação carrega as informações referentes ao perfil do usuário; 3. Para cada item do perfil do usuário: 3.1. *Framework* aplica as configurações de internacionalização; 3.2. *Framework* aplica as demais configurações ao sistema; 4. Aplicação é exibida de acordo com o perfil do usuário.	
Fluxo(s) alternativo(s)	
Não se aplica.	
Fluxo(s) de exceção	
Não se aplica.	
Pós-condições	Aplicação foi exibida seguindo as especificações do perfil do usuário *logado* no sistema.
Regras	1. Ao cadastrar um usuário no sistema, um perfil padrão deve ser gerado para o usuário;
Pontos de extensão	1. **Gerenciar falhas**: se algum tipo de falha ocorrer, então, o fluxo de controle será encaminhado para o caso de uso Gerenciar falhas.

	2. **Manter *log* de atividades**: todas as ações realizadas pelo sistema são incluídas em um *log* de atividades. 3. **Autenticar usuário**: usuário precisa logar no sistema para que aas configurações do perfil sejam carregadas; 4. **Internacionalizar aplicação**: o processo de internacionalização dever ser executado ao carregar o perfil do usuário;

Fonte: Kamyla Estuqui Parrado (2008)

A Tabela 38 apresenta a especificação do caso de uso Gerenciar relatório.

Tabela 38 – Especificação do caso de uso Gerenciar relatórios

Caso de uso	Gerenciar relatórios
Objetivo	Gerenciar a visualização dos relatórios disponibilizados pelas aplicações construídas a partir do *framework*.
Importância	Risco baixo, prioridade baixa.
Ator principal	Usuário da aplicação
Ator(es) secundário(s)	Aplicação
Ativação	Menu ou botões de acesso a relatório.
Pré-condições	Usuário deve estar logado no sistema.
Fluxo principal	

1. Usuário da aplicação clica no menu ou botão de acesso ao relatório desejado;
2. Usuário da aplicação informa os parâmetros necessários;
3. Usuário da aplicação seleciona o formato no qual deseja visualizar o relatório (PDF, RTF, XLS, etc);
4. Aplicação verifica se os parâmetros necessários para exibir o relatório foram preenchidos:
   4.1. Se os parâmetros estirem corretos, então, aplicação direciona o fluxo de controle para o *framework* juntamente com os parâmetros necessários;
   4.2. Senão, enquanto os parâmetros estiverem incorretos, aplicação exibe uma mensagem ao usuário solicitando que este preencha os parâmetros necessários;
5. *Framework* procura pelo relatório que será exibido:
   5.1. Se o relatório for encontrado, então:
      5.1.1. *Framework* envia os parâmetros para o relatório;

5.1.2. *Framework* altera o formato do relatório para o formato escolhido pelo usuário;

5.1.3. Relatório é exibido ao usuário no formato desejado;

6. Senão, se o relatório não for encontrado, então, fluxo de controle é direcionado para o gerenciamento de falhas.

Fluxo(s) alternativo(s)	
Não se aplica.	
Fluxo(s) de exceção	
Não se aplica.	
Pós-condições	Um relatório foi exibido ao usuário
Regras	Não se aplica.
Pontos de extensão	1. **Gerenciar falhas**: se algum tipo de falha ocorrer, então, o fluxo de controle será encaminhado para o caso de uso Gerenciar falhas. 2. **Manter *log* de atividades**: todas as ações realizadas pelo sistema são incluídas em um *log* de atividades.

Fonte: Kamyla Estuqui Parrado (2008)

A Tabela 39 apresenta a especificação do terceiro caso de uso originado do cenário arquitetural *Log* de atividades: Enviar e-mail.

Tabela 39 – Especificação do caso de uso Enviar e-mail

Caso de uso	Enviar e-mail
Objetivo	Permitir à aplicação, ou ao usuário da aplicação enviar e-mails.
Importância	Risco baixo, prioridade baixa.
Ator principal	Aplicação.
Ator(es) secundário(s)	Não se aplica.
Ativação	Ocorrência de algum tipo de falhas.
Pré-condições	Todos os parâmetros necessários para enviar o e-mail devem estar disponíveis.
Fluxo principal	
1. Aplicação verifica se possui todos os parâmetros necessários para efetuar o envio do e-mail: 1.1. Se a aplicação possuir todos os parâmetros, então:	

1.1.1. Aplicação cria um objeto com os parâmetros e envia o objeto para o *framework*;

1.2. Senão, se não possuir todos os parâmetros, então exibe uma mensagem de que não conseguiu enviar o e-mail e informa o motivo;

2. *Framework* lê o objeto recebido e gera um e-mail com as configurações contidas no objeto;
3. *Framework* envia o e-mail;
4. *Framework* retorna um valor para aplicação informando se o e-mail foi enviado com sucesso, ou não.

Fluxo(s) alternativo(s)	
Não se aplica.	
Fluxo(s) de exceção	
Não se aplica.	
Pós-condições	Um e-mail foi enviado.
Regras	1. Parâmetros necessários: e-mail, senha, endereço do servidor SMTP, *flag* para autenticação, *flag* para SSL [31], porta utilizada pelo servidor SMTP, título do e-mail, corpo e-mail e endereço de e-mail dos destinatários.
Pontos de extensão	1. **Gerenciar falhas**: se algum tipo de falha ocorrer, então, o fluxo de controle será encaminhado para o caso de uso Gerenciar falhas. 2. **Manter *log* de atividades**: todas as ações realizadas pelo sistema são incluídas em um *log* de atividades.

Fonte: Kamyla Estuqui Parrado (2008)

3.3.5. Modelo de casos de uso

O objetivo do diagrama de casos de uso é facilitar a visualização das interações entre atores e casos de uso. Os casos de uso são ferramentas utilizadas para discutir requisitos do sistema com os clientes. Portanto, quanto mais fácil for visualizar as funcionalidade do sistema, mais fácil será para clientes e desenvolvedores entrar em acordo sobre as funcionalidades oferecidas pelo sistema.

[31] *Security Sockets Layer*. Protocolo utilizado para garantir a integridade e a privacidade dos dados enviados.

O modelo de casos de uso ilustra graficamente os casos de uso *framework* e os atores que interagem com esses casos de uso. A Figura 15 apresenta o diagrama de casos de uso do *framework.* Os pontos de extensão foram omitidos do diagrama para uma melhor visualização.

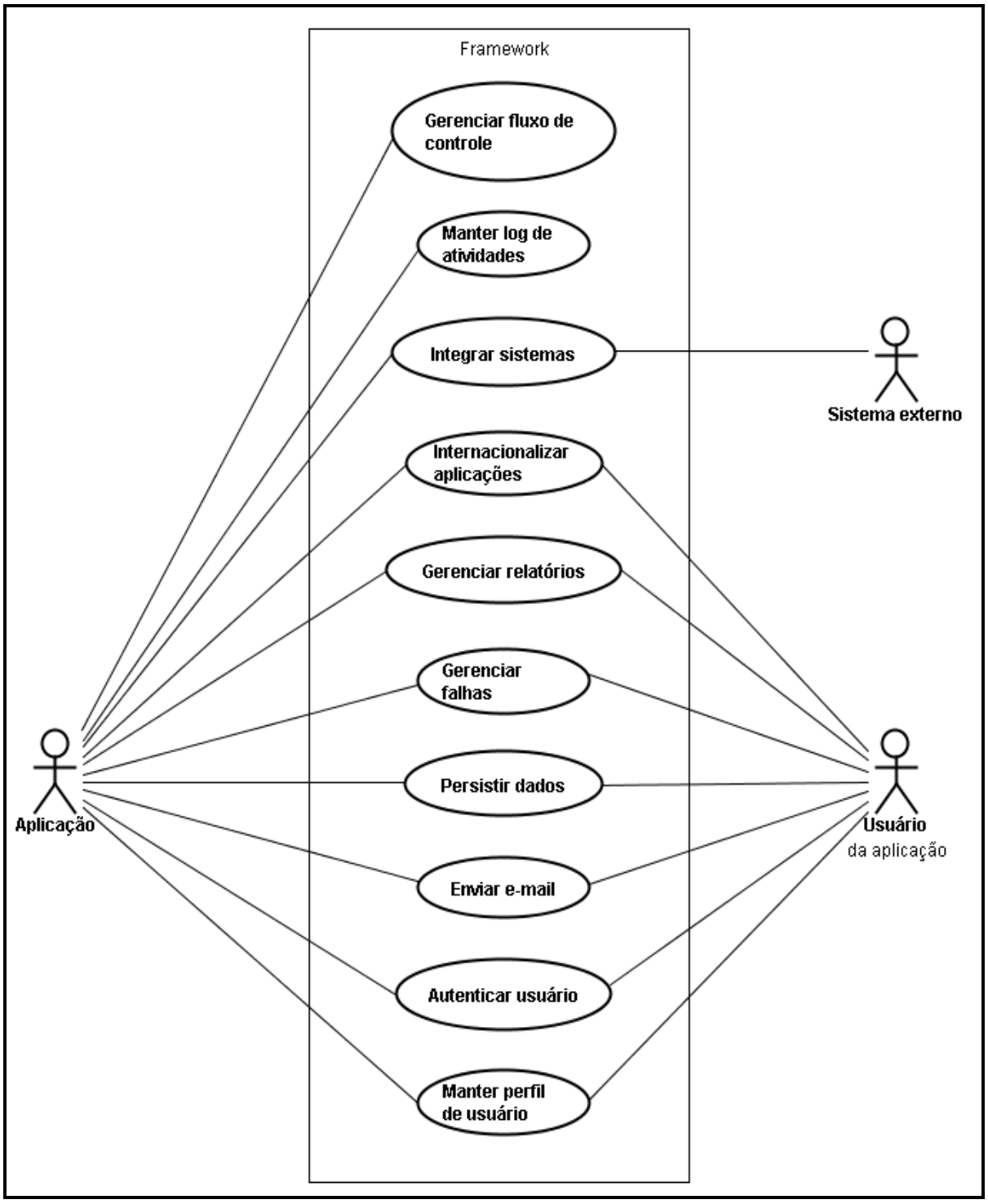

Figura 15 – Modelo de casos de uso
Fonte: Kamyla Estuqui Parrado (2008)

3.3.6. Especificação suplementar dos casos de uso

Esta seção descreve as regras e requisitos não-funcionais que são comuns a todos os casos de uso. Ou seja, são regras e requisitos do *framework* e não de um caso de uso específico. Além dos requisitos já apresentados nos cenários arquiteturais, outros requisitos não-funcionais são de estrema importância para o *framework*:

- **Manutenibilidade**: a evolução do *framework* depende deste requisito. Se o *framework* não for fácil de manter, então, ao invés de inserir novas funcionalidades no *framework*, os desenvolvedores optarão por deixar essas funcionalidades na própria aplicação.
- **Usabilidade**: normalmente a usabilidade está associada à interface gráfica do usuário. No entanto, a usabilidade diz respeito à facilidade de utilização de um sistema. Portanto, a usabilidade é um requisito não-funcional muito importante para um *framework*. Pois, se o *framework* for difícil de entender e utilizar, os desenvolvedores de aplicação vão preferir utilizar soluções alternativas ao *framework*. Dessa forma, o *framework* se tornará obsoleto em pouco tempo.
- **Desempenho**: normalmente aplicações que utilizam *frameworks* tendem a ter seu desempenho afetado pelo *framework*. Ou seja, se o *framework* for lento, o desempenho da aplicação será comprometido. Dessa forma, o desempenho do *framework* deve impactar o mínimo possível no desempenho das aplicações.
- **Flexibilidade**: o *framework* deve ser flexível o suficiente para garantir que possam ser realizadas modificações para suportar novos requisitos.
- **Portabilidade**: as funcionalidades oferecidas pelo *framework*, os recursos (incluindo APIs e outros *frameworks*) utilizados por ele devem ser portáveis às plataformas de hardware e software mais utilizadas do mercado.

3.4. Análise de requisitos

Depois de identificados os requisitos e especificados os casos de uso do *framework*, é necessário avaliar esses requisitos para determinar o que realmente pode ser desenvolvido. E, principalmente, se as soluções propostas no levantamento de requisitos atendem integralmente aos requisitos do *framework*.

Com base nos cenários arquiteturais identificados e na descrição dos estilos arquiteturais feita na fundamentação teórica, foram identificadas algumas arquiteturas que podem ser utilizadas para o desenvolvimento do *framework*.

Para cada arquitetura foi realizada uma análise seguindo a proposta do método de análise arquitetural SAAM. Segundo Mendes (2002), esse método propõe que, para cada cenário, seja determinado se a arquitetura oferece suporte integral às tarefas associadas ao cenário ou, se alguma modificação é necessária e, portanto, a arquitetura oferece suporte parcial.

Para realizar essa análise foi criada uma tabela onde são apresentadas as arquiteturas e os requisitos não-funcionais. Essa tabela é uma adaptação do método SAAM. Para avaliar cada requisito, foi levada em consideração a descrição dos estilos arquiteturais feita na fundamentação teórica deste projeto.

Para cada arquitetura seguiram-se as seguintes regras:

- Se o estilo arquitetural contempla integralmente o requisito em questão, é atribuído o sinal de (+);
- Se o estilo arquitetural não oferece suporte ao requisito não-funcional em questão, então, atribui-se o sinal de (-);
- Se o estilo arquitetural contempla apenas parcialmente o requisito não-funcional em questão, então, atribui-se o sinal (+/-).

A Tabela 40 apresenta a avaliação das arquiteturas candidatas.

Tabela 40 – Avaliação de arquiteturas candidatas

Requisitos arquiteturais	Camadas	Objetos	MVC	SOA
Modularidade	+	+	+	+
Usabilidade	+/-	+	+	+
Flexibilidade	+	+	+	+
Manutenibilidade	+/-	+	+	+
Confiabilidade	-	-	-	-
Disponibilidade	-	-	-	+
Desempenho	-	+/-	+/-	-
Portabilidade	+/-	+/-	+/-	+
Reusabilidade	+/-	+	+	+
Segurança	-	-	-	-
Interoperabilidade	-	-	-	+

Fonte: Kamyla Estuqui Parrado (2008)

A arquitetura em camadas oferece suporte direto aos requisitos de modularidade e flexibilidade. No entanto, a inserção de muitas camadas em uma aplicação pode comprometer a modularidade, o desempenho, a manutenibilidade e a reusabilidade. O desempenho é comprometido porque os dados precisarão passar por muitas camadas antes de serem processados. A modularidade, a manutenibilidade e a reusabilidade são comprometidas quando as camadas de uma aplicação estão fortemente acopladas. Quando isso ocorre, substituir uma camada pode impactar negativamente nas outras camadas. Em alguns casos, modificar uma camada pode se tornar inviável. No entanto, quando as camadas de uma aplicação são bem estruturadas, e o acoplamento entre elas é fraco, os requisitos de modularidade, manutenibilidade e reusabilidade são favorecidos.

A arquitetura orientada a objetos encapsula dados e operações em uma mesma estrutura. Dessa forma os requisitos de modularidade, usabilidade, flexibilidade, manutenibilidade e reusabilidade são favorecidos, pois as alterações realizadas em um objeto serão refletidas no sistema inteiro.

A arquitetura MVC favorece a modularidade usabilidade, flexibilidade, manutenibilidade e reusabilidade das aplicações. Isso ocorre porque existe uma separação entre os objetos de negócio a interface gráfica com o usuário e o tratamento de eventos do sistema. Dessa forma, é possível, por exemplo, substituir toda a interface gráfica de um sistema sem a necessidade de alterar outras partes, como os objetos de negócio.

Os requisitos não-funcionais de confiabilidade, disponibilidade, interoperabilidade desempenho e segurança, dependem de todas as decisões tomadas durante o projeto do sistema. Nenhum estilo arquitetural pode garantir que um sistema será confiável ou terá bom desempenho. No entanto, alguns estilos arquiteturais oferecem um ambiente mais favorável a certos requisitos, como é o caso de SOA e a interoperabilidade.

SOA é uma das arquiteturas mais flexíveis disponíveis hoje. Através da Tabela 40 é possível perceber que SOA suporta a maioria dos requisitos não-funcionais avaliados. Inclusive, é a única que oferece suporte direto a interoperabilidade. No entanto, é importante ressaltar que para obter todos os benefícios de SOA, é necessário que os sistemas desenvolvidos utilizando este paradigma sigam os preceitos impostos pela arquitetura.

Embora um estilo arquitetural ofereça suporte integral a um determinado conjunto de requisitos, não existe garantia de que, na prática, o estilo arquitetural realmente atenderá aos requisitos não-funcionais pré-estabelecidos. Todos os requisitos não-funcionais podem ser comprometidos por outros fatores, como por exemplo, a má estruturação do sistema ou a má utilização de algum tipo de tecnologia.

Com base na análise realizada sobre as arquiteturas candidatas, foram escolhidas as arquiteturas: orientada a objeto e em camadas para o desenvolvimento do *framework*. Para a integração entre sistemas será utilizado SOA. Para a implementação das aplicações construídas a partir do *framework*, serão utilizadas as arquiteturas: orientada a objetos e em camadas. No caso da arquitetura em camadas, quando aplicada às aplicações, utilizará o modelo MVC.

3.5. Design

O design representa a etapa onde às soluções e casos de usos, identificados durante as fases de levantamento e análise de requisitos, são estruturados em um nível mais próximo da implementação.

Nesta etapa são descritos os componentes do *framework* e a forma como estes componentes estão organizados. Além disso, é descrita a maneira como as linguagens de programação *Java* e *Flex* vão se comunicar e, como as aplicações construídas a partir do *framework* devem ser estruturadas.

A estrutura do *framework* foi dividida em duas camadas: *Java* e *Flex*. A maior parte dos casos de uso utiliza a linguagem de programação *Java*. Para a implementação dos casos de uso, é necessário a utilização de funcionalidades adicionais. Alguns exemplos dessas funcionalidades são: leitura/gravação de arquivos, criptografia de dados, geração de arquivos XML, validação de dados, etc. Essas funcionalidades também estão presentes na parte *Java* do *framework*.

A camada *Flex* do *framework* representa os componentes para interface gráfica. Além desses componentes, a camada *Flex* inclui funcionalidades necessárias para controlar os componentes de tela (por exemplo, habilitar e desabilitar campos) e validar os dados inseridos pelos usuários. Os usuários do *framework* podem escolher entre validar os dados em tela, ou enviar os dados para serem validados na camada Java.

3.5.1. Integração entre *Java* e *Flex*

Java e *Flex* são linguagens distintas que precisam se comunicar para que possam ser utilizadas em conjunto. Existem diversas ferramentas que podem ser utilizada para fazer a

comunicação entre *Java* e *Flex*. Algumas dessas ferramentas são: *Adobe LiveCycle*, *Adobe BlazeDS*, *GranitDS*, *OpenAMF*, etc.

Entre essas ferramentas, o *Adobe BlazeDS* se destaca, principalmente, por ser *opensource* e mantido pela *Adobe Systems*. Devido a isso, será utilizado o *Adoble BlazeDS* para fazer a comunicação entre *Java* e *Flex*.

O *BlazeDS* "fornece um conjunto de serviços que permitem conectar uma aplicação cliente a um servidor de dados, e passar dados entre múltiplos clientes conectados ao servidor" (ADOBE SYSTEMS, 2008[a], p.2).

O *BlazeDS* permite acessar objetos remotos através de *HTTPServices*, *Web Services*, *RemoteObject*, etc. A técnica de *RemoteObject* normalmente é utilizada para integrar aplicações *Flex* a aplicações *Java*, no entanto, também pode ser utilizado com PHP e *Microsoft .Net*.

Aplicações desenvolvidas utilizando o *BlazeDS*, são divididas em duas partes:

- ***Client-side***: normalmente uma aplicação desenvolvida em *Flex* ou *AIR*. As aplicações cliente utilizam as funcionalidades disponíveis no servidor. Uma aplicação cliente desenvolvida em *Flex* ou *AIR* costuma ser utilizada apenas como camada de apresentação, ou seja, interface gráfica do usuário.
- ***Server-side***: representa todo o código-fonte escrito em *Java* (ou outra linguagem s*erver-side*) que é executado no servidor.

O *BlazeDS* dispõem de diversas técnicas para integração entre *Flex* e outras linguagens. A técnica escolhida para o *framework* é conhecida como *Remoting Service*. A escolha dessa técnica deve-se a simplicidade e eficiência oferecida pelo *Remoting Service*.

O *Remoting Service* "permite a aplicações cliente acessar métodos de objetos *Java*, sem configurar os objetos como *web services*. Para acessar os objetos *Java*, a aplicação cliente usa o componente *RemoteObject*" (ADOBE SYSTEMS, 2008[a], p.110).

O componente *RemoteObject* permite acessar a lógica de negócio diretamente, no formato nativo em que foi criado, sem que seja necessário transformar os dados que serão enviados/recebidos em XML (ADOBE SYSTEMS, 2008[a])).

Os *RemoteObjects* utilizam *destinations* para acessar as classes *Java*. Cada *destination* representa uma classe. Os *destinations* fornecem uma maneira centralizada de gerenciar os *RemoteObjects*.

3.5.2. Estrutura do *framework* e das aplicações

Como dito anteriormente, o *framework* foi divido em duas camadas: *Java* e *Flex*. Cada camada foi divida em pacotes. Esses pacotes agrupam as classes necessárias para a implementação dos casos de uso.

A camada *Java* contém os seguintes pacotes:

- **Persistência de dados**: agrupas as classes necessárias para manipulação de dados em bancos de dados relacionais.
- **Gerenciamento de falhas**: contém as classes responsáveis pelo gerenciamento de falhas das aplicações construídas a partir do *framework*.
- **Manipulação de arquivos**: contém classes utilizadas para leitura e gravação de arquivos em disco.
- **Segurança**: agrupa as classes necessárias para manter um nível mínimo de segurança nas aplicações construídas a partir do *framework*.
- **Geração de *log* de atividades**: contém as classes responsáveis pelo *log* das atividades das aplicações.
- **Integração**: agrupa as classes responsáveis pela integração entre sistemas.
- **Tratamento e validação de dados**: contém as classes necessárias para validar se os dados informados pelos usuários das aplicações estão corretos. Também compreende classes necessárias para conversão de dados (por exemplo, uma *string* digitada pelo usuário no formato “0,00” deve ser convertida para um número decimal no formato “0.00” para poder ser gravado em banco).
- **Impressão e relatórios**: agrupas as classes responsáveis pelo gerenciamento de relatórios e impressões das aplicações.

A camada *Flex* é composta pelos seguintes pacotes:

- **Componentes gráficos**: agrupas as classes que correspondem aos componentes gráficos utilizados para desenhar as interfaces gráficas das aplicações.
- **Controles de tela**: contém as classes responsáveis pelos controles de tela como habilitar ou desabilitar campos, de acordo com as ações dos usuários das aplicações.
- **Validação de dados**: agrupa as classes utilizadas para validar os dados informados pelos usuários das aplicações.

Entre as camadas *Java* e *Flex* encontra-se o *BlazeDs*, fazendo a comunicação entre as duas linguagens. A Figura 16 ilustra uma representação de alto nível da estrutura do *framework*.

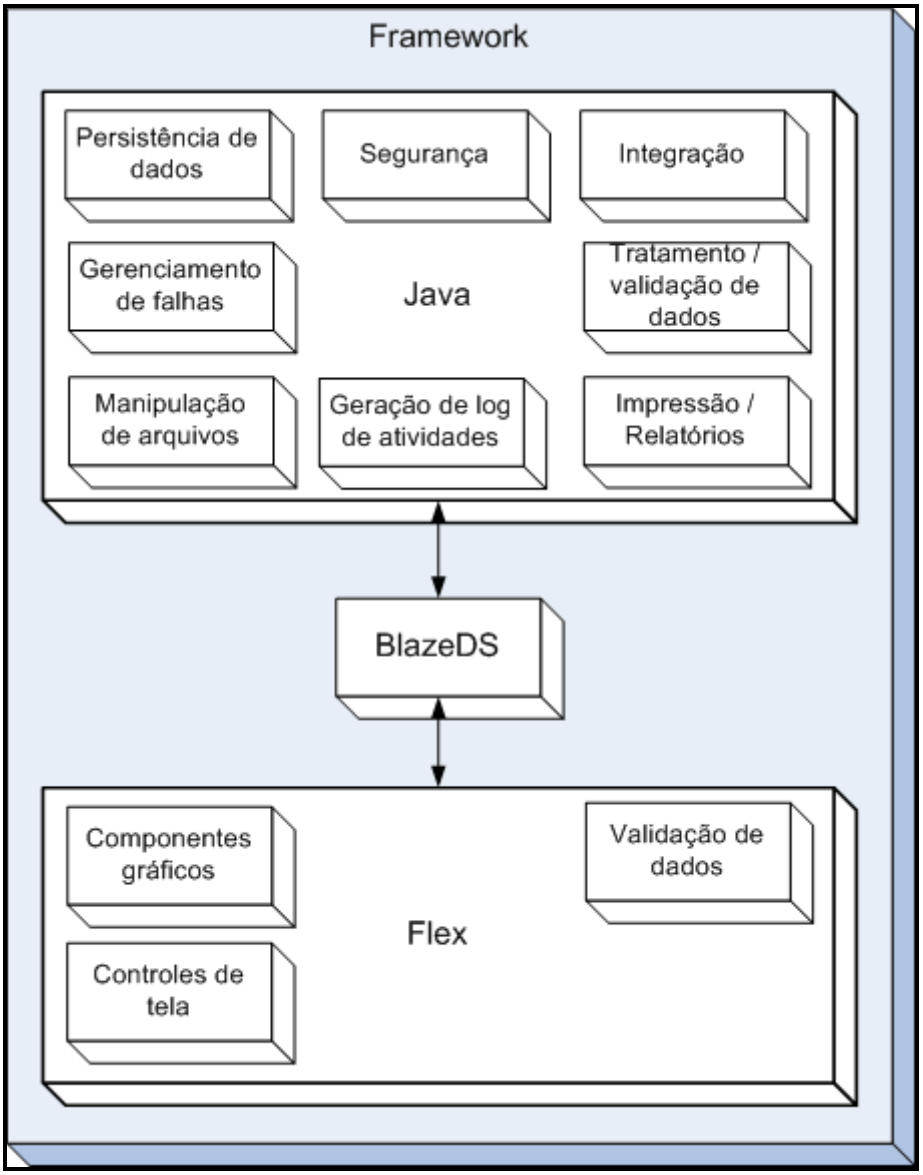

Figura 16 – Representação da estrutura do *framework*
Fonte: Kamyla Estuqui Parrado (2008)

As aplicações construídas a partir do *framework* também devem ser divididas em duas camadas:

- ***Java***: contém as regras de negócio da aplicação em questão.
- ***Flex***: contém a interface gráfica da aplicação em questão. Além disso, também inclui classes espelhos. As classes espelhos são necessárias para representar os objetos de negócios desenvolvidos em *Java* dentro de uma aplicação *Flex*. Ou seja, todos os objetos de negócios escritos em *Java*, devem ter uma classe espelho em *Flex*.

Novamente, entre as camadas (*Java* e *Flex*) encontra-se o *BlazeDS* fazendo a comunicação entre as linguagens.

Tanto as aplicações quanto o *framework* são executados em servidor e, podem utilizar outras APIs e *frameworks* disponíveis. A Figura 17 ilustra uma representação de alto nível da estrutura das aplicações construídas a partir do *framework*.

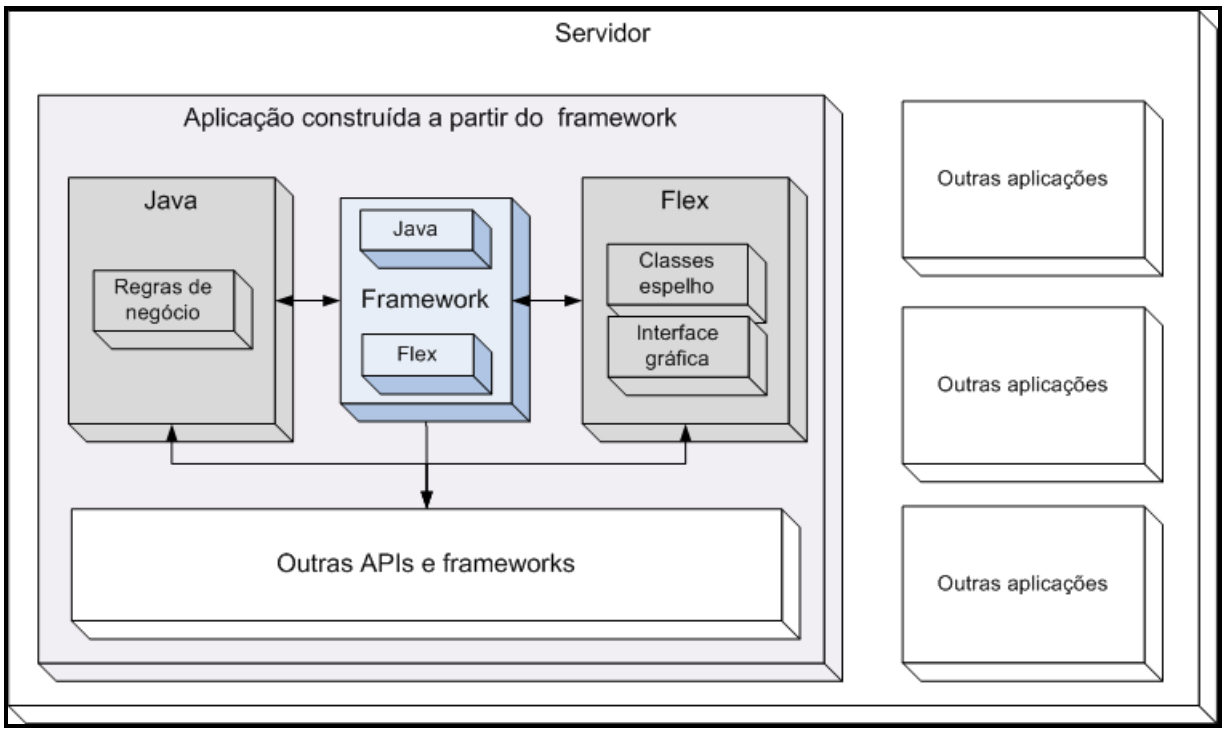

Figura 17 – Representação da estrutura das aplicações construídas a partir do *framework*
Fonte: Kamyla Estuqui Parrado (2008)

3.6. Implementação

A implementação representa a fase onde a especificação de um sistema é transformada em artefatos de software executáveis. Essa seção descreve as ferramentas, técnicas e processos utilizados para construir o *framework*.

3.6.1. Ferramentas e linguagens de programação utilizadas

As ferramentas utilizadas para desenvolver o *framework* são as seguintes:

- ***Eclipse IDE for Java EE Developers* (3.4 – *Ganymed*)**: esta é a principal IDE disponível no mercado para desenvolvimento *Java*. Além de *Java*, a IDE suporta outras linguagens como *Ruby*, *C/C++*, *PHP*, etc. Essa ferramenta é *opensource*.
- ***Flex Builder* (*Plugin for Eclipse*)**: é um *plugin* que permite o desenvolvimento de aplicações *Flex* ou *AIR* utilizando a IDE *Eclipse*. Esta ferramenta é desenvolvida e

comercializada pela *Adobe Systems*, portanto, é uma ferramenta paga. No entanto, a *Adobe* disponibiliza uma licença educacional que permite a estudantes e professores utilizar essa ferramenta sem custo algum, desde que a ferramenta seja utilizada para fins educacionais.

- ***Apache Tomcat***: é um dos servidores de aplicação *Java* para *web* mais populares do mercado. As aplicações construídas a partir do *framework* são executadas utilizando este servidor. A escolha do *Tomcat* se deve ao fato deste servidor já ser utilizado pela empresa. No entanto, as aplicações também podem ser executadas em outros servidores, como *Glassfish*, *Jetty*, *JBoss*, etc.

O *framework* foi desenvolvido utilizando a linguagem *Java* 6 (*update* 10) e o *Flex* SDK (3.1). As mesmas ferramentas utilizadas para desenvolver o *framework*, também podem ser utilizadas para desenvolver as aplicações construídas a partir do *framework*.

3.6.2. Comunicação com o banco de dados

As aplicações construídas a partir do *framework* utilizam um mecanismo de persistência automatizado. Esse mecanismo foi desenvolvido utilizando o *EntityManager* fornecido pela JPA.

De acordo com Burke e Monson-Haefel (2007, p. 42),

> O *EntityManager* é o serviço central para todas as ações de persistência. [...]. O *EntityManager* gerencia o mapeamento O/R entre um conjunto fixo de classes de entidade e uma origem de dados subjacente. Ele fornece APIs para criar consultas, localizar objetos, sincronizar objetos e inserir objetos no banco de dados. Ele também pode fornecer armazenamento em *cache* e gerenciar a interação entre uma entidade e os serviços transacionais em um ambiente *Java* EE como o JTA. O *EntityManager* está bem integrado ao *Java* EE e ao EJB, mas não está limitado a esse ambiente, ele pode ser utilizado em programas *Java* simples.

Apesar de todas as características presentes no *EntityManager*, a JPA exige que seja utilizado um provedor de persistência, que é o verdadeiro responsável por realizar a persistência. Para o *framework* foi utilizado o *Hibernate*. No entanto, é possível substituir o *Hibernate* por qualquer outro provedor de persistência que siga a especificação da JPA.

O *framework* fornece um implementação padrão para acesso e manipulação de dados. Essa implementação foi construída utilizando os padrões de projeto DAO[32], *Singleton* e

[32] *Data Access Object*. É um padrão de projeto, cujo objetivo é separar a lógica utilizada para acesso e manipulação de dados, da lógica de negócio de uma aplicação.

Factory. Embora seja uma implementação padrão, é possível estender as funcionalidades disponíveis para adaptá-las às necessidades específicas de cada aplicação.

A persistência de dados é composta por uma interface (*IBaseDao*), que fornece os métodos básicos para manipulação de dados e duas classes:

- ***Dao***: uma classe genérica que implementa a interface *IBaseDao*, fornecendo a implementação padrão para persistência. Essa classe recebe em tempo de execução o tipo de objeto que está sendo manipulado e, pode ser estendida pelas aplicações.
- ***DaoFactory***: é a classe responsável pela instanciação da classe *Dao*. Utiliza-se o padrão de projeto *Singleton* para garantir que existe apenas uma instância da classe *Dao* para cada tipo de objeto tratado. Essa classe precisa ser estendida pelas aplicações construídas a partir do *framework.*

A JPA, normalmente, é utilizada em conjunto com POJOs (*Plain Old Java Object*), que nada mais são do que classes *Java* que contém os atributos que representam um objeto e os métodos de acesso a esses atributos (*getters* e *setters*).

A JPA disponibiliza anotações que são utilizadas para indicar que uma classe será persistida em banco e para configurar como a classe será persistida. Algumas dessas anotações são:

- ***@Entity***: indica que a classe será persistida em banco;
- ***@Id***: indica que determinado atributo de uma classe será a chave primária dessa classe quando persistida;
- ***@GeneratedValue***: essa anotação é usada em conjunto com a anotação *@Id* para indicar como será a geração da chave primária.
- ***@Column***: oferece opções para personalizar a forma como um atributo de uma classe será mapeado para um campo de uma tabela do banco de dados;
- ***@Temporal***: indica que um campo representa uma data ou hora.

Existe um conjunto de outras anotações, tanto da JPA quanto de suas implementações, que podem ser utilizadas para mapear relacionamentos entre classes, indicar a multiplicidade, etc. Cada anotação possui propriedades que podem ser utilizadas para personalizar a forma como uma classe será persistida. Outras anotações da JPA muito utilizadas são: *@ManyToMany*, *@OneToOne*, *@OneToMany*, *@ManyToOne*, *@Cascade*, etc.

A Figura 18 ilustra um POJO com as anotações da JPA. Esse POJO representa um objeto de negócio de uma aplicação construída a partir do *framework*.

```
 1 package com.br.java.framework.entities;
 2
 3 import java.util.Calendar;
13
14 @Entity
15 public class Contato {
16
17     @Id
18     @GeneratedValue(strategy = GenerationType.IDENTITY)
19     private int oid;
20     @Column(nullable = false)
21     private String nome;
22     private String endereco;
23     private String bairro;
24     private String cidade;
25     private String estado;
26     private String cep;
27     private String telefone;
28     private String celular;
29     private String email;
30     @Temporal(TemporalType.DATE)
31     private Calendar dataCadastro;
32
33     public Contato() {
34     }
35
36     public String getNome() {
37         return nome;
38     }
39
40     public void setNome(String nome) {
41         this.nome = nome;
42     }
43     // outros getters e setters...
```

Figura 18 – Exemplo de POJO com anotações da JPA
Fonte: Kamyla Estuqui Parrado (2008)

Além das anotações, é necessário utilizar um arquivo, conhecido como *persistence.xml* para configurar a persistência de dados. O objetivo deste arquivo é configurar o funcionamento do provedor de persistência. Entre outras informações, esse arquivo contém os dados necessários para que a aplicação se conecte com o banco de dados. Este arquivo deve ficar em um diretório chamado META-INF dentro do projeto da aplicação.

Esse arquivo deve ser incluído em todas as aplicações construídas a partir do *framework*. O conteúdo desse arquivo pode variar, dependendo da implementação escolhida e das necessidades da aplicação. A Figura 19 ilustra um exemplo de configuração para um arquivo *persistence.xml*.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence>
    <persistence-unit name="persistencecontext">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <class>com.br.java.framework.entities.Contato</class>
        <properties>
            <property name="hibernate.connection.driver_class">
                com.mysql.jdbc.Driver
            </property>
            <property name="hibernate.connection.url">
                jdbc:mysql://localhost:3306/agines_novo
            </property>
            <property name="hibernate.connection.username">
                root
            </property>
            <property name="hibernate.connection.password">
                root
            </property>
            <property name="hibernate.dialect">
                org.hibernate.dialect.MySQLDialect
            </property>
            <property name="hibernate.hbm2ddl.auto">update</property>
        </properties>
    </persistence-unit>
</persistence>
```

Figura 19 – Exemplo de *persistence.xml*
Fonte: Kamyla Estuqui Parrado (2008)

Algumas *tags* do arquivo *persistence.xml* são:

- ***<provider></provider>***: representa o nome da classe da implementação da JPA escolhida, ou seja indica qual é o provedor de persistência;
- ***<class></class>***: identificam as classes que serão persistidas em banco, para cada classe deve ser inserida uma *tag* como esta;
- ***hibernate.connection.driver_class***: representa à classe do *driver* JDBC que será utilizado, normalmente esse informação pode ser obtida através da documentação do *driver*;
- ***hibernate.connection.url***: representa uma "*string* de conexão", ou seja, o caminho para o banco de dados;
- ***hibernate.connection.username***: indica o nome de usuário utilizado para acessar o banco de dados;
- ***hibernate.connection.password***: representa a senha utilizada para acessar o banco de dados;
- ***hibernate.dialect***: indica o dialeto utilizado para que a implementação da JPA se comunique com o banco de dados;

- ***hibernate.hbm2ddl.auto***: indica que a implementação da JPA pode criar automaticamente *schemas* de bancos de dados.

3.6.3. Comunicação entre *Java* e *Flex*

Todo projeto *web* criado em *Java* contém um *Deployment Descriptor*, conhecido como *web.xml*. Esse arquivo está localizado dentro do diretório *WEB-INF* do projeto e, contêm as configurações do projeto, como parâmetros de inicialização, mapeamento de *servlets*, etc. A Figura 20 ilustra um exemplo de *Deployment Descriptor*.

```
1<?xml version="1.0" encoding="UTF-8"?>
2<!DOCTYPE web-app PUBLIC "-//Sun Microsystems,
3Inc.//DTD Web Application 2.3//EN"
4    "http://java.sun.com/dtd/web-app_2_3.dtd">
5<web-app>
6    <display-name>TestesFramework</display-name>
7    <!-- MessageBroker Servlet -->
8    <servlet>
9        <servlet-name>MessageBrokerServlet</servlet-name>
10       <display-name>MessageBrokerServlet</display-name>
11       <servlet-class>
12           flex.messaging.MessageBrokerServlet
13       </servlet-class>
14       <init-param>
15           <param-name>services.configuration.file</param-name>
16           <param-value>/WEB-INF/flex/services-config.xml</param-value>
17       </init-param>
18       <init-param>
19           <param-name>flex.write.path</param-name>
20           <param-value>/WEB-INF/flex</param-value>
21       </init-param>
22       <load-on-startup>1</load-on-startup>
23   </servlet>
24   <servlet-mapping>
25       <servlet-name>MessageBrokerServlet</servlet-name>
26       <url-pattern>/messagebroker/*</url-pattern>
27   </servlet-mapping>
28   <welcome-file-list>
29       <welcome-file>index.html</welcome-file>
30       <welcome-file>index.htm</welcome-file>
31       <welcome-file>index.jsp</welcome-file>
32   </welcome-file-list>
33</web-app>
```

Figura 20 – Exemplo de *Deployment Descriptor*
Fonte: Kamyla Estuqui Parrado (2008)

Este é um exemplo simplificado de um *Deployment Descriptor*, onde se encontra:

- O mapeamento para o *servlet* responsável pela comunicação entre *Java* e *Flex*;
- Dois parâmetros de inicialização indicando a localização de arquivos utilizados para configurar o *BlazeDS*;
- A lista de *welcome files* (possíveis páginas iniciais que serão acessadas quando um usuário acessar o projeto).

O *Deployment Descriptor* do projeto indica o caminho de dois arquivos utilizados para configurar o BlazeDS: *services-config.xml* e *remoting-config.xml*.

O arquivo *services-config.xml* define os canais utilizados por uma aplicação cliente para se comunicar com um servidor. Esse arquivo também contém configurações de segurança, *log*, etc. Qualquer alteração nesse arquivo requer que o projeto seja compilado novamente.

O arquivo *remoting-config.xml* é utilizado para configurar os *destinations* dos *Remoting Services*, ou seja, este arquivo faz o mapeamento das classes *Java* que serão acessadas via *RemoteObject*. Dentro desse arquivo encontram-se *tags* XML chamadas de *destination*. Essas *tags*, que possuem um ID (representando o nome do *destination*), e o endereço onde se encontra a classe que será acessada através do *BlazeDS*. Para cada classe é necessário criar uma *tag* como a apresentada na Figura 21.

```
1<destination id="contatoServiceHibernateJpa">
2    <properties>
3        <source>
4            com.br.java.framework.dao.ContatoHibernateJpaDao
5        </source>
6    </properties>
7</destination>
```

Figura 21 – Exemplo de *destination*
Fonte: Kamyla Estuqui Parrado (2008)

Nem todos os dados que trafegam entre *Java* e *Flex* são tipos padronizados, como *String*, *Interger*, *Double*, *Boolean*, etc. Normalmente é necessário transferir objetos como Cliente, Produto, etc. Para que seja possível que *Java* e *Flex* entendam esses objetos, é necessário que, para cada objeto de negócio criado em *Java*, seja escrita uma classe espelho em *Flex*. A Figura 22 apresenta a classe espelho do POJO apresentado na Figura 18.

```
1  package com.br.flex.framework.entities
2  {
3      [RemoteClass(alias="com.br.java.framework.entities.Contato")]
4      [Bindable]
5      public class Contato
6      {
7          public function Contato()
8          {
9          }
10
11         public var oid : Number;
12         public var nome : String;
13         public var endereco : String;
14         public var bairro : String;
15         public var cidade : String;
16         public var estado : String;
17         public var cep : String;
18         public var telefone : String;
19         public var celular : String;
20         public var email : String;
21
22     }
23 }
```

Figura 22 – Exemplo de classe espelho
Fonte: Kamyla Estuqui Parrado (2008)

Essa classe espelho nada mais é do que uma cópia de uma classe *Java*, escrita em *Flex*. É possível perceber que a classe espelho possui uma *metadata* chamada *RemoteClass*. Essa *metadata* indica qual classe do modelo de domínio escrito em *Java* é representada pela classe espelho. Ou seja, essa *metadata* faz o mapeamento direto da classe.

3.7. Testes

O *framework* será testado utilizando os seguintes tipos de testes:

- **Testes de unidade**: este tipo de teste será automatizado através da utilização das ferramentas *EasyMock* e *JUnit*.
- **Testes de integração**: o objetivo desses testes é garantir que os componentes desenvolvidos são capazes de trabalhar em conjunto. Esses testes serão automatizados através de ferramenta *TestNG*.
- **Testes de estresse**: através da simulação de situações extremas, que exijam muito processamento, será avaliado o comportamento do *framework*. O objetivo desses

testes é identificar os pontos do *framework* que podem comprometer o desempenho das aplicações.

- **Teste de regressão**: para cada modificação realizada no *framework*, todo o conjunto de testes realizados anteriormente deverá ser refeito.
- **Testes de sistema e aceitação**: após a finalização da construção dos componentes do *framework*, será desenvolvido um aplicativo de referência. O objetivo da construção desse aplicativo é:
    - Identificar e corrigir o maior número possível de falhas do *framework*, antes que este entre em produção;
    - Verificar a aceitação do *framework* entre os desenvolvedores de aplicação.

O *JUnit* é um *framework opensource* que permite a criação de testes de unidade em aplicações *Java*. As IDEs mais populares do mercado, como *Eclipse* e *Netbeans*, fornecem suporte direto à criação de testes unitários com este *framework*.

De acordo com Veloso (2008), o *core*[33] do *JUnit* é composto pelas classes *Assert*, *TestResult*, *Test*, *TestListener*, *TestCase*, *TestSuite* e *BaseTestRunner*. No entanto, esse autor também afirma que a estrutura principal do *framework* é composta por:

- ***TestCase***: classe que deve ser estendida por todas as classes que implementem casos de testes utilizando o *JUnit*;
- ***TestSuite***: representa um conjuntos de casos de testes (classes que estendem *TestCase*);
- ***BaseTestTunner***: é a classe base utilizada por todos os testes.

Existem objetos em um sistema que possuem dependências, ou seja, para que um objeto realize determinada função, ele precisa que outro objeto exista. Ao realizar testes unitários, nem sempre as dependências de todos os objetos estão disponíveis. Quando isso ocorre, os desenvolvedores de testes podem fazer uso de objetos *mock*.

Araujo e Quintela (2008) definem objetos *mock* como "objetos 'falsos' que simulam o comportamento de uma classe ou objeto 'real'". Para esses autores, os objetos *mock* auxiliam os desenvolvedores a se manter focados nos casos de testes, e não no código necessário para que um teste unitário possa ser realizado adequadamente. O objetivo do *framework EasyMock* é justamente facilitar a criação de objetos *mock*. Os *frameworks EasyMock* e *JUnit* trabalham em conjunto. O *EasyMock* utiliza o ambiente fornecido pelo *JUnit* para realizar os testes.

[33] O *core* representa as classes principais de um sistema, ou seja, as classes que fornecem os principais serviços de um sistema.

O *framework TestNG* é considerado o sucessor do *JUnit*. O *TestNG* apresenta muitas das características presentes no *JUnit*, no entanto, esse *framework* também introduz uma série de novos conceitos que facilitam o desenvolvimento de testes.

De acordo com Rocha (2008, p.80),

> O *framework* contém diversas anotações referentes à configuração dos testes, que permitem a execução de métodos específicos antes e depois da execução dos testes, suítes, grupos, métodos e classes. Para completar, o *TestNG* emprega conceitos avançados de testes como dependência entre testes, agrupamento, dados de testes, métodos de testes com parâmetros, configuração de execução de testes através de arquivos XML, etc. O *framework* possui integração com diversas IDEs e um utilitário que permite a conversão de casos de testes criados no *JUnit*.

Através das definições sobre as ferramentas utilizadas para realizar os testes, é possível perceber que esses *frameworks* apresentam características em comum. No entanto, optou-se pela utilização dos três *frameworks*, e não apenas um deles, para avaliar qual dessas ferramentas se adapta melhor às necessidades do *framework*.

Os casos de testes são utilizados para guiar o processo de testes. Esses casos de testes representam testes unitários que serão automatizados através das ferramentas descritas anteriormente.

Para o caso de uso Persistir dados foram identificados os casos de testes apresentados na Tabela 41.

Tabela 41 – Casos de testes do caso de uso Persistir dados

Caso de teste	Descrição
Validar campos obrigatórios	Verifica se todos os campos obrigatórios para a criação de um objeto foram preenchidos.
Validar tipo de dado	Verifica se os valores informados (pelo usuário da aplicação ou por sistemas externos) correspondem aos tipos de dados esperados pelos atributos do objeto que será persistido.
Verificar persistência	Para cada operação realizada pelo caso de uso verifica se o tipo de retorno corresponde descrito na Tabela 42

Fonte: Kamyla Estuqui Parrado (2008)

Os casos de testes Validar campos obrigatórios e Validar tipo de dado representação pré-condições para a execução correta do caso de uso Persistir dados. A Tabela 42 apresenta os resultados esperados após a execução dos casos de testes especificados na Tabela 41.

Tabela 42 – Tabela de decisão para o caso de uso Persistir dados

Causa	Efeito
Caso de teste: validar campos obrigatórios	
Atributo é obrigatório e valor é = “”[34]	Inválido
Atributo é obrigatório e valor é != “”	Válido
Caso de teste: validar tipo de dado	
Tido do atributo = Data E valor >= 01/01/1900 E <= 01/01/2100	Válido
Tido do atributo = Data E valor (< 01/01/1900 OU > 01/01/2100)	Inválido
Tido do atributo = Data E valor = “@#$ 123 abc” [35]	Inválido
Tido do atributo = Caractere E valor = “@#$ 123 abc”	Válido
Tipo do atributo = Numérico E Aceita valores negativos E valor < 0	Válido
Tipo do atributo = Numérico E Aceita valores negativos E valor >= 0	Válido
Tipo do atributo = Numérico E Aceita valores negativos E valor = “”	Inválido
Tipo do atributo = Numérico E Aceita valores negativos E valor = “@#$ 123 abc”	Inválido
Tipo do atributo = Numérico E Não aceita valores negativos E valor < 0	Inválido
Tipo do atributo = Numérico E Não aceita valores negativos E valor >= 0	Válido
Tipo do atributo = Numérico E Não aceita valores negativos E valor = “”	Inválido
Tipo do atributo = Numérico E Não aceita valores negativos E valor = “@#$ 123 abc”	Inválido
Caso de teste: Validar persistência	
Operação = criar objeto E (retorno = *null* OU OID = 0)	Inválido
Operação = editar objeto E (retorno = *null*)	Inválido
Operação = remover objeto E (retorno = 0)	Inválido
Operação = consultar objeto(s) E (retorno = *null*)	Inválido
Operação = criar objeto E (retorno != *null* OU OID != 0)	Válido
Operação = editar objeto E (retorno != *null*)	Válido
Operação = remover objeto E (retorno = 1)	Válido
Operação = consultar objeto(s) E (retorno = coleção de objetos)	Válido

Fonte: Kamyla Estuqui Parrado (2008)

[34] “” (aspas duplas) indicam que não foi informado valor algum.
[35] Esses caracteres representam valores alfanuméricos e caracteres especiais.

Para o caso de uso Internacionalizar aplicações foi identificado o caso de testes apresentado na Tabela 43.

Tabela 43 – Casos de testes do caso de uso Internacionalizar aplicações

Caso de teste	Descrição
Verificar se existe arquivo de internacionalização	Verifica se existe um arquivo com as configurações necessárias para internacionalizar as aplicações.

Fonte: Kamyla Estuqui Parrado (2008)

A Tabela 44 apresenta os resultados esperados após a execução dos casos de testes do caso de uso Internacionalizar aplicações.

Tabela 44 – Tabela de decisão para o caso de uso Internacionalizar aplicações

Causa	Efeito
Caso de teste: verificar se existe arquivo de internacionalização	
Arquivo encontrado	Válido
Arquivo não encontrado	Inválido
Arquivo encontrado E Ilegível	Inválido
Arquivo encontrado E Não contém todas as informações necessárias	Inválido

Fonte: Kamyla Estuqui Parrado (2008)

Para o caso de uso Manter *log* de atividades foram identificados os casos de testes apresentados na Tabela 45.

Tabela 45 – Casos de testes do caso de uso Manter *log* de atividades

Caso de teste	Descrição
Validar criação do arquivo de *log*	Verifica se o arquivo onde serão inseridas as informações de *log* foi gerado corretamente.
Verificar o conteúdo padrão do arquivo de *log*	Verifica que o arquivo de *log* contém todas as informações descritas no caso de uso Manter *log* de atividades.
Validar gravação do arquivo de *log*	Verifica se o arquivo de *log* foi gravado corretamente no banco de dados.
Verificar exclusão do arquivo de *log*	Verifica se o arquivo foi excluído após ser gravado no banco de dados.

Fonte: Kamyla Estuqui Parrado (2008)

A Tabela 46 apresenta os resultados esperados após a execução dos casos de testes para o caso de uso Manter *log* de atividades.

Tabela 46 – Tabela de decisão para o caso de uso Manter *log* de atividades

Causa	Efeito
Caso de teste: Validar criação do arquivo de *log*	
Arquivo gerado	Válido
Arquivo não gerado	Inválido
Caso de teste: Verificar o conteúdo padrão do arquivo de *log*	
Arquivo contém todas as informações	Válido
Arquivo não contém ao menos uma informação obrigatória	Inválido
Caso de teste: Validar gravação do arquivo de *log*	
Arquivo gravado no banco de dados	Válido
Arquivo não gravado no banco de dados	Inválido
Caso de teste: Verificar exclusão do arquivo de *log*	
Arquivo gravado no banco E Excluído	Válido
Arquivo gravado no banco E Não excluído	Inválido
Arquivo não gravado no banco E Excluído	Inválido
Arquivo não gravado no banco e Não excluído	Inválido

Fonte: Kamyla Estuqui Parrado (2008)

A Tabela 47 apresenta os casos de testes do caso de uso Autenticar usuário.

Tabela 47 – Casos de testes do caso de uso Autenticar usuário

Caso de teste	Descrição
Validar *login*	Verifica se existe um usuário cadastrado no sistema com o *login* informado.
Validar senha	Verifica se a senha informada pelo usuário esta correta.
Validar usuário	Verifica se apenas usuários com *login* e senha corretos conseguem acessar o sistema.

Fonte: Kamyla Estuqui Parrado (2008)

A Tabela 48 apresenta os resultados esperados após a execução dos casos de testes do caso de uso Autenticar usuário.

Tabela 48 – Tabela de decisão para o caso de uso Autenticar usuário

Causa	Efeito
Caso de teste: Validar *login*	
Tamanho do *login* >= 8 caracteres	Válido
Tamanho do *login* < 8 caracteres	Inválido
Caso de teste: Validar senha	
Tamanho da senha >= 6 caracteres	Válido
Tamanho da senha < 6 caracteres	Inválido
Caso de teste: Validar usuário	
Senha Incorreta E *Login* correto E Usuário consegue acessar o sistema	Inválido
Login correto E Senha incorreta E Usuário consegue acessar os sistema	Inválido
Login incorreto E Senha incorreta E Usuário consegue acessar o sistema	Inválido
Login incorreto E Senha incorreta E Usuário não consegue acessar o sistema	Válido
Login correto E Senha correta E Usuário consegue acessar o sistema	Válido

Fonte: Kamyla Estuqui Parrado (2008)

A Tabela 49 apresenta os casos de testes para o caso de uso Gerenciar relatórios.

Tabela 49 – Casos de testes do caso de uso Gerenciar relatórios

Caso de teste	Descrição
Validar parâmetros	Verifica se todos os parâmetros necessários para exibir o relatório foram informados.
Verificar arquivo do relatório	Verifica se o arquivo *.jasper* existe.

Fonte: Kamyla Estuqui Parrado (2008)

A Tabela 50 apresenta os resultados esperados após a execução dos casos de testes do caso de uso Gerenciar relatórios.

Tabela 50 – Tabela de decisão para o caso de uso Gerenciar relatórios

Causa	Efeito
Caso de teste: Validar parâmetros	
Todos os parâmetros preenchidos	Válido
Pelo menos um parâmetro não preenchido	Inválido
Caso de teste: Verificar arquivo do relatório	
Arquivo não encontrado	Inválido

Arquivo encontrado	Válido
Arquivo encontrado E Relatório não exibido	Inválido

Fonte: Kamyla Estuqui Parrado (2008)

3.8. Implantação

A implantação do *framework* se iniciará após a finalização dos testes necessários para garantir que todos os componentes estão funcionando adequadamente. O primeiro projeto que utilizará o *framework* será a migração do sistema *Agines Network* para uma nova estrutura, seguindo a arquitetura fornecida pelo *framework* desenvolvido.

Para a implantação do *framework* é necessário que todas as ferramentas listadas na Tabela 51 estejam disponíveis.

Tabela 51 – Ferramentas necessárias para implantação do *framework*

Ferramenta	Descrição
Eclipse	IDE utilizada para desenvolver o *framework* e as aplicações construídas a partir do *framework*.
Flex Builder (Eclipse Plugin)	*Plugin* que adiciona as funcionalidades presentes no *Flex Builder* ao *Eclipse*. Essa ferramenta é necessária para o desenvolvimento de aplicações escritas em *Flex* ou *AIR*.
Apache Tomcat	Servidor utilizado para distribuir as aplicações construídas a partir do *framework*.
Flex SDK	*Kit* de desenvolvimento necessário para construir aplicações utilizando o *Flex*.
Java SDK	*Kit* de desenvolvido necessário para construir aplicações utilizando a plataforma *Java*.

Fonte: Kamyla Estuqui Parrado (2008)

Não será necessário alterar o hardware utilizado atualmente, pois, este, suporta adequadamente todas as ferramentas listadas na Tabela 51.

3.9. Manutenção

A manutenção é uma atividade de extrema importância para qualquer *framework*. A manutenção ocorrerá das seguintes formas:

- **Corretiva**: necessária para corrigir os problemas que venham a surgir com a utilização *framework*;
- **Adaptativa**: os requisitos do *framework* e das aplicações construídas a partir dele vão mudar e, com essas mudanças, será necessário adaptar o *framework* a nova realidade.

A manutenção adaptativa está diretamente associada à evolução do *framework*. Os novos requisitos darão origem a novos casos de uso, ou a mudanças na estrutura dos casos de uso identificados. Será necessário manter um histórico das alterações feitas no *framework* para que essas alterações não tenham impacto negativo nas aplicações já construídas a partir do *framework.*

Assim como será necessário que o *framework* seja adaptado a novos requisitos, também pode ser necessária a criação de novas funcionalidades para que o *framework* não se torne obsoleto.

3.10. Documentação

A documentação de um *framework* é uma das partes mais importantes do seu desenvolvimento. É necessário que existam documentos que ensinem aos desenvolvedores da aplicação como utilizar o *framework*, quais são seus pontos de extensão, quais são as funcionalidades disponíveis, etc. Além disso, para facilitar a manutenção do *framework* é necessário documentar todas as funcionalidade implementadas.

A documentação *framework* será feita das seguintes maneiras:

- ***Javadoc*** [36] : para documentar o código-fonte do *framework*, ou seja, como as funcionalidade foram implementadas;
- ***Cookbooks***: são tutoriais que contém exemplos de como o *framework* pode ser utilizado;
- **Cartões *hot spot***: serão utilizados para documentar os pontos de extensão do *framework* e qual o comportamento esperado para cada ponto de extensão;
- **Diagramas UML e especificação de casos de uso**: serão utilizados para documentar o projeto do *framework*, ou seja, é uma documentação de alto-nível que pode ser utilizada para manter um histórico sobre as mudanças ocorridas no *framework*.

[36] Ferramenta disponível na plataforma *Java* para criação de arquivos HTML contendo a documentação do código-fonte das aplicações escritas em *Java*.

CONSIDERAÇÕES FINAIS

Quando os primeiros softwares começaram a ser escritos, ninguém podia imaginar o papel que este assumiria na sociedade. Hoje o software está presente em todos os lugares e disponível para uma variedade muito grande de dispositivos.

Conforme a importância dos softwares foi aumentando, foi necessário o desenvolvimento de tecnologias que pudessem facilitar o seu desenvolvimento, tornando este processo mais rápido, fácil e menos trabalhoso. Algumas dessas tecnologias estão voltadas para domínios específicos, como o desenvolvimento de aplicações *web*, outros para domínios tecnológicos, como sistemas orientados a objetos.

Os *frameworks* são uma tecnologia promissora, que auxiliam não apenas a desenvolver aplicações, mas, principalmente, a reutilizar implementação, análise e *design* de software. Além disso, os *frameworks* aumentam a qualidade e a confiabilidade que as aplicações construídas a partir dele apresentam. Isso ocorre porque a estrutura disponibilizada pelo *framework* já foi amplamente testada, devido a sua utilização em diversos projetos.

Durante o desenvolvimento deste projeto, os conceitos apresentados na fundamentação teórica foram de suma importância. Esses conceitos auxiliaram no desenvolvimento de uma solução que segue os princípios propostos pela engenharia de software.

Como citado na fundamentação teórica, um *framework* está em constante evolução. Devido a isso, não foi possível identificar todos os componentes que um *framework* deve possuir de uma única vez. Dessa forma, inicialmente, foram desenvolvidos apenas os componentes essenciais para o desenvolvimento dos softwares da empresa.

O desenvolvimento de *frameworks* exige que diversos desafios sejam vencidos. Alguns desses desafios foram encontrados durante a execução deste projeto. O principal deles foi à alta curva de aprendizado, pois, foi necessária uma ampla pesquisa e análise para encontrar as soluções que foram utilizadas para desenvolver o *framework*. Essa análise foi necessária para que o *framework* fosse desenvolvido utilizando as melhores técnicas e ferramentas disponíveis no mercado.

Um ponto que contribuiu negativamente para o desenvolvimento de *frameworks* foi à inexistência de metodologias e ferramentas que ofereçam suporte a esta atividade. Nenhuma das três metodologias apresentadas neste projeto define mecanismos para modelagem e documentação do *framework*. Isso se torna um problema, uma vez que a modelagem e a documentação são pontos cruciais para a manutenção e evolução do *framework*.

A utilização de casos de uso e diagramas UML não foi suficiente para descrever todos os pontos da estrutura de um *framework*. O objetivo dos casos de uso é descrever processos de negócio, eles não devem incluir detalhes sobre tecnologia. No entanto, grande parte dos componentes (casos de uso) do *framework* estão diretamente associados a detalhes técnicos/tecnológicos.

A UML dispõe de um perfil conhecido com UML-F que visa auxiliar a modelagem de *frameworks*. No entanto, devido a pouca documentação sobre este perfil, não foi possível utilizá-lo para modelar o *framework*.

Outro problema foi o fato da carga horária do estágio ser insuficiente para desenvolver um *framework*. Isso se deve, principalmente, a curva de aprendizado necessária para desenvolver o *framework*.

Durante o levantamento de requisitos, os principais cenários arquiteturais foram identificados e descritos. Esses cenários foram baseados na utilização e desenvolvimento do sistema *Agines Network*. As deficiências encontradas neste sistema foram os principais agentes motivadores para o desenvolvimento do *framework*. Portanto, os cenários arquiteturais e casos de uso descritos visam atender as necessidades mais importantes desse sistema.

Durante a análise de requisitos foram analisadas as arquiteturas que estão sendo utilizadas para definir e implementar a estrutura do *framework*. Para cada arquitetura foram apresentados os pontos positivos e negativos. As arquiteturas escolhidas para o *framework* foram: em camadas e orientadas a objetos. Para a integração entre sistemas foi escolhida SOA. As aplicações construídas a partir do *framework* utilizam arquiteturas em camada (MVC) e orientada a objetos.

Durante fase de design, foram descritos os componentes do *framework* e a forma como esses componentes foram organizados. O *framework* foi dividido em duas camadas: *Java* e *Flex*. Entre essas camadas encontra-se o *BlazeDS*, que é o responsável pela comunicação entre essas duas linguagens.

Atualmente, o *framework* encontra-se parcialmente implementado. Os casos de uso Persistir dados, Enviar e-mail, Autenticar usuários e Gerenciar relatórios, são os que estão em estágio mais avançado de desenvolvimento. Para os casos de uso parcialmente implementados, foram realizados testes unitários seguindo os casos de testes apresentados na seção de testes deste projeto.

Devido a complexidade envolvida com a criação de algoritmos para inversão de controle, o caso de uso Gerenciar fluxo de controle só será implementado após a conclusão dos demais casos de uso.

Implementar SOA requer um amplo estudo sobre as reais necessidades dos sistemas que utilizarão este estilo arquitetural. Embora um dos casos de uso deste projeto tenha como objetivo integrar sistemas através da implementação de SOA utilizando *web services*, não faz parte desse projeto realizar o estudo necessário para a implementação de SOA. Devido a isso, o caso de uso Integrar sistemas só será implementado após a realização desse estudo e implementação dos demais casos de suo.

Apesar dos desafios encontrados, após a conclusão deste projeto de estágio, o desenvolvimento do *framework* continuará. E, o mais importante, é que mesmo se deparando com diversos problemas, a empresa continuará investindo no desenvolvimento de soluções reutilizáveis, o que inclui a finalização e o aperfeiçoamento do *framework* proposto por este projeto.

REFERÊNCIAS

ADOBE SYSTEMS. **BlazeDS developer guide**. s.l. Adobe Systems, 2008[a]. Disponível em: <http://livedocs.adobe.com/blazeds/1/blazeds_devguide/blazeds_devguide.pdf>. Acesso em: 1 set. 2008.

_____. **Flex 3 developer guide**. s.l. Adobe Systems, 2008[b]. Disponível em: <http://livedocs.adobe.com/blazeds/1/blazeds_devguide/ devguide_flex3.pdf>. Acesso em: 1 set. 2008.

_____. **Página da Adobe Systems sobre internacionalização de aplicações no Flex 3**. Disponível em: <http://labs.adobe.com/wiki/index.php/Flex_3:Feature_Introductions:_Runtime_Localization>. Acesso em: 1 ago. 2008[c].

_____. **Página oficial do produto Adobe Flex**. Disponível em: <http://www.adobe.com /products/flex/>. Acesso em: 13 ago. 2008[d].

AHMED, Khawar Zaman; UMRYSH, Cary E. **Desenvolvendo aplicações comerciais em Java com J2EE e UML**. Rio de Janeiro: Ciência Moderna Ltda, 2002.

ANDRADE, Gilberto Keller; AUDY, Jorge Luis Nicolas; CIDRAL, Alexandre. **Fundamentos de sistemas de informação**. Porto Alegre: Bookman, 2005.

ARAUJO, Marco Antônio Pereira; QUINTELA, Bárbara de Melo. Testes com objetos mock: utilizando o framework EasyMock para testes unitários de aplicações Java. **Engenharia de Software Magazine**. v. 1, n. 6, p. 42, out. 2008.

BAIROS, Alexandre; PEREIRA, Bruno. Web services REST: uma abordagem prática. **Java Magazine**. v. 1, n 56, p. 26, maio 2008.

BARRETO, Celso Gomes. **Agregando frameworks de infra-estrutura em uma arquitetura baseada em componentes**: um estudo de caso no ambiente AulaNet. 2006. 210f. Dissertação (Mestrado em Informática) – Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro. Disponível em <http://www2.dbd.pucrio.br/pergamum/tesesabertas/0410823_06_pretextual.pdf>. Acesso em: 10 mar. 2008.

BAUER, Christian; KING, Gavin. **Java persistence com Hibernate**. Rio de Janeiro: Editora Ciência Moderna, 2007.

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

BIRT. **Página oficial da ferramenta BIRT**. Disponível em: <http://www.eclipse.org/ birt/phoenix/>. Acesso em: 18 set. 2008.

BLAHA, Michael; RUMBAUGH, James. **Modelagem e projetos baseados em objetos**. 2.ed. Rio de Janeiro: Campus, 2006.

BOOCH, Grady; JACOBSON, Ivar; RUMBAUGH, James. **UML Guia do usuário**. 3.ed. Rio de Janeiro: Editora Campus, 2005.

_____. UML **Guia do usuário**. 2.ed. Rio de Janeiro: Editora Campus, 2000.

BOOTH, David *et al*. **Web services architecture**. s.l. W3C, 2004. Disponível em: <http://www.w3.org:80/TR/2004/NOTE-ws-arch-20040211/ >. Acesso em: 23 set. 2008.

BUBACK, Silvano; PEREIRA, Bruno. JSR-311 e Jersey: web services REST no Java EE. **Java Magazine**. v.1, n. 60, ago. 2008.

BURKE, Bill; MONSON-HAEFEL, Richard. **Enterprise JavaBeans 3.0**. São Paulo: Pearson Prentice Hall, 2007.

CHIAVENATO, Idalberto. Teoria de Sistemas. *In*: _____. **Introdução à Teoria Geral da Administração**. 6.ed. Rio de Janeiro: Editora Campus, 2000. cap. 17, p.543 – 576.

CORREIA, Carlos Henrique; TAFNER, Malcon Anderson. **Análise orientada a objetos**. Florianópolis: Visual Books, 2001.

CRYSTAL REPORTS. **Página oficial do produto CrystalReports**. Disponível em <http://www.businessobjects.com/product/catalog/crystalreports/>. Acesso em: 18 set. 2008.

DATE, C. J. **Introdução a sistemas de Banco de Dados**. 7.ed. Rio de Janeiro: Campus, 2000.

DEITEL, Harvey M. **Java como programar**. 6.ed. São Paulo: Prentice-Hall, 2005.

DIAS, Arilo Cláudio. Introdução a teste de software. **Engenharia de Software Magazine**. v. 1, n. 1, p. 54, mar. 2008.

FAYAD, Mohamed; SCHMIDT, Douglas C. **Object-oriented application frameworks**. Communications of the ACM, s.l, vol. 40, no. 10, out. 1997. Disponível em: <http://www.cse.wustl.edu /~schmidt/CACM-frameworks.html >. Acesso em: 24 jul. 2008.

FERREIRA, Alessandro. **Frameworks e padrões de projeto**. Disponível em <http://www.devmedia.com.br/articles/viewcomp.asp?comp=1111&hl=>. Acesso em: 10 mar. 2008.

FILEDING, Thomas. **Architectural styles and the design of network-based software architectures**. 2000. 180f. Dissertation (Doctor of Philosophy in Information and Computer Science) – University of California., Irvine. Disponível em: <http://www.ics.uci.edu/~fielding /pubs/dissertation/fielding_dissertation.pdf>. Acesso em: 24 set. 2008.

FOWLER, Martin. **UML essencial**: um breve guia para linguagem-padrão de modelagem de objetos. 3.ed. Porto Alegre: Bookman, 2005.

FURGERI, Sérgio. **Java 2**: ensino didático. São Paulo: Editora Érica Ltda, 2002.

FURLAN, José Davi. **Modelagem de objetos através da UML**. São Paulo: Macron Books, 1998.

GAMMA, Erich et al. **Padrões de projeto**: soluções reutilizáveis de software orientado a objetos. Porto Alegre: Bookman, 2000

HARRINGTON, Jan. **Projetos de bancos de dados relacionais**: teoria e prática. 2.ed. Rio de Janeiro: Campus, 2002.

HIBERNATE. **Página oficial do projeto Hibernate**. Disponível em: <http://hibernate.org/>. Acesso em: 04 ago. 2008.

IBATIS. **Página oficial do projeto Apache iBATIS**. Disponível em: <http://ibatis.apache .org/>. Acesso em: 04 ago. 2008.

JASPER FORGE. **Página oficial da API JasperReports**. Disponível em: <http://jasperforg e.org/plugins/project/project_home.php?group_id=102>. Acesso em: 18 set. 2008[a].

_____. **Página oficial da ferramenta iReport**. Disponível em: <http://jasperfo rge.org/ plugins/project/project_home.php?group_id=83>. Acesso em: 18 set. 2008[b].

JAVA COMMUNITY PROCESS. **Página da Java Specification Requeste (JSR) para a API de Logging do Java**. Disponível em <http://www.jcp.org/en/jsr/detail?id=47>. Acesso em: 18 set. 2008.

KAZOUN Chafic; LOTT Joey. **Programming Flex**. Sebastopol: O'Reilly, 2007.

KODALI, Raghu R. **What is a service-oriented architecture?** s.l. JavaWorld, 2005. Disponível em: <http://www.javaworld.com/javaworld/jw-06-2005/jw-0613-soa.html>. Acesso em: 23 set. 2008.

KORTH, Henry; SILBERSCHATZ, Abrahan; SUDARSHAN, S. **Sistemas de bancos de dados**. 3.ed. São Paulo: Makron Books, 1999.

KUCHANA, Partha. **Software architecture**: design patterns in Java. Boca Raton: Auerbach Publications, 2004.

LARMAN, Craig. **Utilizando UML e padrões**: uma introdução à análise e ao projeto orientados a objetos. Porto Alegre: Bookman, 2002.

LAUDON, Kenneth C; LAUDON, Jane Price. **Gerenciamento de sistemas de informação**. 3.ed. Rio de Janeiro: LTC – Livros Técnicos e Científicos S.A., 2001.

_____. **Sistemas de informação**. 4.ed. Rio de Janeiro: LTC – Livros Técnicos e Científicos S.A., 1999.

LIMA, Adilson da Silva. **UML 2.0**: do requisito à solução. São Paulo: Érica, 2005.

LOG4J. **Página oficial do projeto Apache Log4J**. Disponível em: <http://logging.apache.org/>. Acesso em: 18 ago. 2008.

MELO, Ana Cristina. **Desenvolvendo aplicações com UML**. Rio de Janeiro: Brasport, 2002.

MENDES, Antonio. **Arquitetura de software**: desenvolvimento orientado para arquitetura. Rio de Janeiro: Campus, 2002.

_____. Usabilidade de software: a importância da usabilidade no desenvolvimento de sistemas interativos. **Engenharia de Software Magazine**. v. 1, n. 5, p. 24, set. 2008.

MICROSOFT. **Página oficial do produto Microsoft Silverlight**. Disponível em: <http://www.microsoft.com/SILVERLIGHT/>. Acesso em: 13 ago. 2008.

MOREIRA, Trayahú R; RIOS, Emerson. **Projeto e engenharia de software**: teste de software. Rio de Janeiro: Alta Books, 2003.

O'REILLY, Tim. **What is web 2.0**: design patterns and business models for the next generation of software. Disponível em: <http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html>. Acesso em: 7 jul. 2008.

O'BRIEN, James A. **Sistemas de Informação e as decisões gerenciais na era da Internet**. São Paulo: Saraiva, 2002.

OMG. **The Common Object Request Broker**: architecture and specification. s.l. OMG, 1999. Disponível em: <http://www.omg.org/docs/formal/99-10-07.pdf>. Acesso em: 19 set. 2008.

OPENLASZLO. **Página oficial do projeto OpenLaszlo**. Disponível em: <http://www.openlaszlo.org/>. Acesso em: 13 ago. 2008.

ORACLE. **Página oficial do produto Oracle TopLink**. Disponível em: <http://www.oracle.com/technology/products/ias/toplink/index.html>. Acesso em: 04 ago. 2008.

PEDRYCZ, Witold; PETERS James F. **Engenharia de software**: teoria e prática. Rio de Janeiro: Campus, 2001.

PEREIRA, Bruno. Web services WS-*: uma abordagem prática. **Java Magazine**. v. 1, n 55, p. 24, abr. 2008.

PLATT, Michael. Web 2.0 na empresa. **The Architecture Journal**. v.1, n.12, p.2, Jul. 2007. Disponível em: <http://download.microsoft.com/download/a/b/2/ab24766b-fbc4-43d9-954b-c3504372e0a1/aj12_po.zip>. Acesso em: 10 mar. 2008.

PRESSMAN, Roger S. **Engenharia de Software**. 6.ed. São Paulo: MacGraw-Hill, 2006.

PREVAYLER. **Página oficial do projeto Prevayler**. Disponível em: <http://www.prevayler.org/>. Acesso em: 04 ago. 2008.

REYNOLDS, George W; STAIR, Ralph M. **Princípios de sistemas de informação**: uma abordagem gerencial. 4.ed. Rio de Janeiro: LTP – Livros Técnicos e Científicos Editora S.A., 2002.

REZENDE, Denis Alcides. **Engenharia de software e sistemas de informação**. 3.ed. Rio de Janeiro: Brasport, 2005.

ROCHA, André Dantas. Testes avançados com o TestNG: utilizando boas práticas para realizar testes de forma eficaz. **Java Magazine**. v. 1, n. 62, p. 72, out. 2008.

SANTOS, Guilherme Nascimento Pate. **Introduzindo variabilidade no desenvolvimento de sistemas multi-agentes**. 2007. 110f. Dissertação (Mestrado em Informática) – Pontifícia Universidade Católica do Rio de Janeiro – PUC-Rio, Rio de Janeiro. Disponível em: <http://www.maxwell.lambda.ele.puc-rio.br/cgibin/PRG_0599.EXE/10397_3.PDF?NrOcoSis=33353&CdLinPrg=pt>. Acesso em: 06 maio 2008.

SCHWEBEL, Samuel Cristhian. **Frasedare**: framework orientado a objetos para segurança de dados em repouso. 2005. 165f. Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Santa Catarina, Florianópolis. Disponível em: <http://www.tede.ufsc.br/teses/PGCC0738.pdf>. Acesso em: 10 mar. 2008

SCOTT, Kendall. **O processo unificado explicado**. Porto Alegre: Bookman, 2003.

SILVA, Elaine Quintino da. **Um framework baseado e componentes para desenvolvimento de aplicações web e um processo de instanciação associado**. 2006. 167f. Tese (Doutorado em Ciências – Ciência de Computação e Matemática Computacional) – Universidade de São Paulo – USP, São Paulo. Disponível em: <http://www.teses.usp.br/teses/disponiveis/55/55134/tde-26092006-152652/>. Acesso em: 06 maio 2008.

SILVA, Ricardo Pereira. **Suporte ao desenvolvimento e uso de frameworks de componentes**. 2000. 262f. Tese (Doutorado em Ciência da Computação) – Universidade Federal do Rio Grande do Sul, Porto Alegre. Disponível em: <http://www.inf.ufsc.br/~download/tese.pdf>. Acesso em: 10 mar. 2008.

SOARES, Diogo Castro Veloso. **Testes de unidade com JUnit**. Disponível em: <http://www.devmedia.com.br/ articles/viewcomp.asp?comp=4637&hl=>. Acesso em: 20 nov. 2008.

SOMMERVILLE, Ian. **Engenharia de software**. 6.ed. São Paulo: Addison Wesley, 2003.

SOUZA, Cesar Alexandre de. **Sistemas integrados de gestão empresarial**: estudos de casos de implementação de sistemas ERP. 2000. 306f. Dissertação (Mestrado em Administração) – Universidade de São Paulo, São Paulo. Disponível em: <http://www.teses.usp.br/teses/ disponiveis/12/12133/tde-19012002-123639/>. Acesso em: 12 dez. 2008.

STAIR, Ralph M. **Princípios de sistemas de informação: uma abordagem gerencial**. Rio de Janeiro: LTP – Livros Técnicos e Científicos Editora S.A., 1996.

SUN MICROSYSTEMS. **Página da Sun Microsystems sobre internacionalização em aplicações Java**. Disponível em: <http://java.sun.com/javase/ technologies/core/basic/intl/>. Acesso em: 19 set. 2008.

TALIGENT. **Building object-oriented frameworks**. s.l. Taligent, 1994.Disponível em: <http://lhcb-comp.web.cern.ch/lhcb-comp/Components/postscript/buildingoo.pdf>. Acesso em: 23 jul. 2008.

W3C. **Página oficial dos padrões propostos pelo W3C para internacionalizar aplicações web**. Disponível em: <http://www.w3.org/International>. Acesso em: 16 set. 2008.

WASLAWICK, Raul Sidnei. **Análise e projeto de sistemas de informação orientados a objetos**. 2.ed. Rio de Janeiro: Elsevier, 2004.

ZANCUL, Eduardo de Senzi. **Análise da aplicabilidade de um sistema ERP no processo de desenvolvimento de produtos**. 2000. 242f. Dissertação (Mestrado em Engenharia de Produção) – Universidade de São Paulo, São Carlos. Disponível em: <http://www.teses.usp.br/teses/disponiveis/18/18140/tde-11102001-114028/>. Acesso em: 12 dez. 2008.

GLOSSÁRIO

ABAS – *Attribute-Based Architecture Style*: ferramenta utilizada para compreender qualitativa e quantitativamente os atributos de qualidade de um estilo arquitetural.

ACD – *Architectural Context Diagram*: diagrama utilizado para modelar o contexto arquitetural de um sistema.

ADL – *Architectural Description Language*: linguagens que disponibilizam a sintaxe e a semântica necessária para descrever a arquitetura de um sistema.

AIR – *Adobe Integrated Runtime*: ferramenta que permite desenvolver *Rich Internet Applications* que podem ser executadas no desktop, ou seja, sem a necessidade de utilizar um navegador de Internet.

API – *Application Program Interface*: conjunto de funções que podem ser utilizadas por uma aplicação. A utilização de APIs simplifica o desenvolvimento de software, uma vez que essas APIs disponibilizam uma implementação de funcionalidades que são requeridas por muitas aplicações.

Arquitetura de software: estuda a organização global dos sistemas de software e os relacionamentos existentes entre os subsistemas, conectores e componentes de um sistema.

Arquivos .properties: são arquivos de propriedades onde o conteúdo é apresentado no formato chave=valor.

ATAM – *Architecture Tradeoff Analysis Method*: método de análise arquitetural utilizado, principalmente, para identificar riscos. Ou seja, identificar decisões para as quais não são conhecidas todas as conseqüências.

Cartões *hot spot*: são cartões utilizados para documentar os pontos de extensão de um *framework.*

CASE – *Computer Aided Software Engineering*: ferramentas que oferecem suporte as atividades de desenvolvimento de software.

CBSE – *Component-Based Software Engineering*: também conhecida como desenvolvimento baseado em componentes, enfatiza a construção de software utilizando componentes reutilizáveis.

Cenários de qualidade / cenários arquiteturais: são exemplos de interações entre usuários e o sistema. A construção de cenários de qualidade permite simular situações de interação entre os usuários e o sistema. Dessa forma é possível avaliar se o sistema atende aos requisitos arquiteturais especificados e, principalmente, atende as necessidades dos usuários. O tipo de cenário desenvolvido depende do tipo de sistema ao qual o cenário está associado.

Componente de software: é uma unidade de software desenvolvida para fins específicos e amplamente testado. Esses componentes devem ser úteis, adaptáveis, portáveis e, principalmente, reutilizáveis. O objetivo principal dos componentes de software é reutilizar código escrito em qualquer linguagem e que possa ser executado em qualquer plataforma.

CORBA – *Common Object Request Broker Architecture*: estilo arquitetural criado pela OMG. Tem o objetivo de efetuar troca de objetos entre sistemas distribuídos.

COTS – *Commercial Off-The-Shelf*: são produtos de prateleira, fornecidos por terceiros (fabricantes), que oferecem funcionalidades mais amplas que os sistemas especializados desenvolvidos para uma organização específica.

CRM – *Customer Relationship Management*: ferramentas que disponibilizam uma série de funcionalidades de apoio a vendas, marketing e serviços aos clientes

CRUD – *Create, Read, Update, Delete*: são as quatro operações básicas realizadas em bancos de dados relacionais: criar, ler, atualizar, deletar.

DDL – *Data-Definition Language*: são linguagens utilizadas para definir as tabelas utilizadas pelo banco de dados relacionais.

Disponibilidade: diz respeito à capacidade de um sistema estar operacional sempre que um usuário tentar utilizá-lo.

DML – *Data-Manipulation Language*: são linguagens utilizadas para consultar, atualizar, inserir ou remover dados de um banco de dados relacional.

EJB – *Enterprise Java Beans*: tecnologia pertencente à plataforma *Java*, mais especificamente ao *Java* EE (*Enterprise Edition*). Foi projetada para gerenciar automaticamente muitos dos serviços necessários para o desenvolvimento de aplicações corporativas.

EJB QL – *Enterprise Java Beans Query Language*: linguagem de consulta semelhante ao SQL, porém adaptada para trabalhar com objetos *Java*, ao invés de esquemas relacionais.

ERP – *Enterprise Resource Planning*: sistemas integrados utilizados pelas organizações para controlar todas as suas operações.

Escalabilidade: se refere à capacidade de um sistema suportar um grande número de novos componentes, que não haviam sido previstos no escopo original do sistema, sem afetar negativamente os componentes desenvolvidos.

Estilo arquitetural: representa uma categoria de sistema. Um estilo arquitetural engloba um conjunto de componentes e os conectores utilizados por estes componentes, quais são as restrições de comunicação entre os componentes, etc.

Framework: software parcialmente implementado que pode ser estendido por aplicações, dessa forma, determinando a arquitetura das aplicações construídas a partir dele. É composto por um conjunto de interfaces e classes concretas e abstratas que colaboram entre si para atingir um determinado objetivo.

Frozen spot: são os pontos de um *framework* que não podem ser alterados pelas aplicações construídas a partir dele. Ou seja, os *frozen spots* representam as estruturas reutilizáveis de um *framework.*

GRASP – *General Responsibility Assignment Software Patterns*: são padrões de projeto que descrevem princípios para a atribuição de responsabilidades a objetos.

Hot spot: são os pontos de um *framework* que podem (ou devem) ser customizados pelas aplicações construídas a partir dele.

HQL – *Hibernate Query Language*: é um dialeto SQL utilizado pelo *Hibernate*. Embora sua sintaxe se pareça com SQL, o HQL é totalmente orientado a objetos.

HTTP – *Hypertext Transfer Protocol*: protocolo utilizado para transmissão de dados através da Internet.

i18n – *Internationalization*: é o processo através do qual uma aplicação desenvolvida em um determinado país é preparada para ser utilizada em outros países.

IDE – *Integrated Development Environment*: conjunto de ferramentas que oferecem suporte ao desenvolvimento de software. Normalmente incluem editores de código, depuradores, etc.

IEEE – *Institute of Electrical and Electronics Engineers*: organização profissional sem fins lucrativos que tem como objetivo disseminar conhecimento nas áreas de engenharia elétrica, eletrônica e computação.

Interoperabilidade: é a capacidade de sistemas distintos conseguirem se comunicar através de um conjunto de interfaces previamente definidas.

IoC – *Inversion of Control*: técnica que permite que um *framework* assuma o controle de uma aplicação.

JAR – *Java Archive*: formato de arquivo que contém um conjunto de classes *Java* compiladas que podem representar um aplicativo ou uma API.

Javadoc: ferramenta disponível na plataforma *Java* que permite gerar arquivos HTML contendo a documentação do código fonte de uma aplicação.

JDBC – *Java DataBase Connectivity*: API disponível na plataforma *Java* que fornece funcionalidades para acesso a banco de dados.

JDK – *Java Development Kit*: contém as ferramentas básicas necessárias para desenvolver aplicativos em *Java*.

JPA – *Java Persistence* API: API de persistência da plataforma *Java* que oferece um mecanismo padrão para persistência de dados.

JRE – *Java Runtime Environment*: é o ambiente de execução *Java*. É composto pela JVM e pelas bibliotecas necessárias para executar aplicativos *Java*.

JVM – *Java Virtual Machine*: é a responsável pela execução de programas escritos em *Java* em qualquer plataforma. A JVM faz uma ponte entre o software e o sistema operacional. Dessa forma, um programa escrito em *Java* não tem acesso direto à máquina onde está sendo executado. A máquina virtual é uma camada extra entre a aplicação e o sistema operacional, responsável pela tradução do programa escrito em *Java* em linguagem específica da plataforma onde está sendo executado.

l10n – *Localization*: refere-se à personalização de uma aplicação relacionada a: formatos numéricos, de data e de hora; uso de moeda; texto e gráficos contendo referências a objetos, ações ou idéias que, em determinada cultura, podem estar sujeitos à interpretação incorreta; exigências legais diversas, entre outros.

Manutenibilidade: refere-se à facilidade de adicionar, atualizar ou remover funcionalidades em um sistema. Quanto mais fácil for manter um sistema, maior é o seu grau de manutenibilidade.

MER – Modelo Entidade-Relacionamento: compreende um conjunto de notações que permitem aos usuários/desenvolvedores modelar o banco de dados utilizado por uma aplicação.

Métodos *hook*: são métodos com uma implementação padrão que podem ser redefinidas nas subclasses que estendem a classe que contém estes métodos.

Modelo de ciclo de vida de software: representam as atividades e artefatos produzidos por estas atividades, e as interações que ocorrem durante o ciclo de vida.

Modularidade: é capacidade de dividir um sistema em partes menores, denominadas módulos. A modularidade permite lidar com detalhes de cada módulo isoladamente e, também, lidar com as características comuns aos módulos e as relações existentes entre eles de modo a integrá-lo em um sistema.

MTTF – *Mean Time to Failure*: medida de tempo entre as falhas observadas em um sistema. Oferece um indicativo de quanto tempo o sistema permanecerá operacional antes que uma falha aconteça.

MVC – *Model, View, Controller*: estilo arquitetural que divide uma aplicação em três camadas: *model* (representa os objetos de negócio, o modelo de domínio do sistema); *view* (representa a interface gráfica do sistema); *controller* (representa o controle de eventos que ocorre no sistema).

OMG – *Object Management Group*: organização internacional responsável pela padronização de diversas tecnologias, incluindo CORBA e UML.

OO – Orientação a objetos: é uma técnica utilizada para modelar, analisar, projetar e desenvolver software em termos de objetos. Ou seja, a orientação a objetos utiliza um conjunto de entidades que possuem características e comportamentos próprios, tais como objetos do mundo real, como telefones, produtos, pessoas, etc.

Opensource: são projetos com distribuição livre, incluem não apenas os arquivos binários necessários para a execução do software, mas também o código fonte.

ORB – *Object Request Broker*: representa a base para construir aplicações distribuídas e para a interoperabilidade entre elas utilizando o estilo CORBA. Permite que objetos recebam e enviem requisições e resposta em ambientes distribuídos. O ORB é a parte mais importante do CORBA. O ORB está entre o cliente e o objeto. É o ORB que aceita uma requisição, envia a requisição para o objeto e, quando a resposta está disponível, envia para o cliente.

ORM – *Object-Relational Mapping*: representa o conjunto de técnicas e ferramentas utilizadas para fazer o mapeamento dos objetos de aplicações Orientadas a Objetos para entidades dos bancos de dados relacionais.

Padrão arquitetural: define uma abordagem específica para tratar de algumas características comportamentais do sistema. Padrões arquiteturais podem ser utilizados em conjunto com um estilo arquitetural, definindo a estrutura global de um sistema.

Padrões de projeto: capturam soluções que foram desenvolvidas e aperfeiçoadas ao longo do tempo. Esses padrões refletem a experiência a os esforços dos desenvolvedores por maior reutilização e flexibilidade. Padrões de projeto tornam mais fácil a reutilização de projetos e arquiteturas bem-sucedidas, pois ajudam a escolher alternativas de projeto que tornam um sistema reutilizável e a evitar alternativas que comprometam a reutilização.

Persistência de dados: tem como objetivo guardar os valores que estão sendo manipulados por um sistema, para que estes valores possam ser recuperados posteriormente. É possível salvar esses dados utilizando arquivos, porém, a abordagem mais comum é salvar os dados em bancos de dados.

PHP – *Hypertext Preprocessor*: linguagem de programação utilizada para desenvolver aplicações para o ambiente *web*.

POJO – *Plain Old Java Object*: são classes *Java* que possuem um método construtor vazio e um conjunto de atributos privados com seus respectivos métodos de acesso (*getters* e *setters*).

Ponto de extensão: representam os pontos de um *framework* que podem ser modificados, ou seja, estendido, pelas aplicações construídas a partir dele. Também são conhecidos como *hot spots*.

Portabilidade: diz respeito à facilidade de transferir um sistema de uma plataforma computacional para outra. Sistemas de software são portáveis quando podem ser executados em diversos tipos de ambiente sem que sejam necessárias grandes mudanças no sistema.

Processo de software: é um conjunto de atividades com *feedback*, realizadas com a finalidade de produzir e manter os sistemas de software. Esse conjunto de atividades produz uma série de artefatos que resultam em um programa executável.

QA – *Quality Assurance*: é composta por um conjunto de atividades que definem a estrutura necessária para garantir a qualidade de software. O processo de QA, também, define padrões que devem ser aplicados ao processo de desenvolvimento de software.

RAD – *Rapid Application Development*: modelo de desenvolvimento de software incremental que enfatiza um ciclo de desenvolvimento curto. Este modelo pode ser considerado como uma adaptação do modelo em cascata, onde o desenvolvimento rápido é alcançado através do uso de uma abordagem de construção baseada em componentes.

Reengenharia de software: técnica que propõe a re-implementação de sistemas legados para facilitar a manutenção dos mesmos.

Requisitos arquiteturais: conjunto de requisitos utilizados para determinar qual o melhor estilo arquitetural para um determinado sistema.

REST – *Representational State Transfer*: representa uma abstração dos elementos da arquitetura de sistemas de hipermídia distribuída. REST ignora os detalhes da implementação dos componentes e a sintaxe dos protocolos para se focar nas regras dos componentes e nas restrições sob suas interações com outros componentes.

Reusabilidade: capacidade que um sistema, ou componente, possui de ser utilizado em diversos sistemas sem precisar ser modificado.

RIA – *Rich Internet Application*: conceito inovador para desenvolvimento de aplicações *web*. Sistemas que são desenvolvidos utilizando este conceito unem as funcionalidades disponíveis em aplicativos *desktops* com as facilidades oferecidas pelo ambiente *web*.

RMI – *Remote Method Invocation*: método desenvolvido para permitir que objetos ativos em uma máquina virtual *Java* interajam com objetos ativos em outras máquinas virtuais *Java*.

SAAM – *Software Architecture Analysis Method*: método de análise arquitetural projetado para auxiliar a comparação entre alternativas arquiteturais.

SDK – *Software Development Kit*: conjunto de ferramentas que permitem desenvolvimento de aplicação para uma determinada plataforma.

SGBD – Sistema Gerenciador de Bancos de Dados: fornecem um conjunto de ferramentas que facilitam a manipulação das informações contidas em um banco de dados. Os SGBDs representam uma camada entre os dados físicos e as aplicações ou usuários que utilizam esses dados.

Sistema legado: são sistemas que incorporam as modificações feitas durante muitos anos em um software. Um sistema legado foi desenvolvido para atender às necessidades antigas do cliente e, normalmente, foi implementado utilizando uma tecnologia antiga.

SOA – *Service Oriented Architecture*: estilo arquitetural que visa garantir a interoperabilidade entre sistemas.

SOAP – *Simple Object Access Protocol*: protocolo utilizado para o envio/recebimento de mensagens com *web services* WS-*. O SOAP representa a camada de transporte entre a aplicação consumidora e os serviços disponibilizados pelos *web services*.

SQL – *Structured Query Language*: linguagem de definição e manipulação de dados padrão utilizada pela maioria dos bancos de dados relacionais.

SSL – *Secure Sockets Layer*: protocolo utilizado para garantir a integridade e a privacidade dos dados enviados através da Internet.

SWOT – *Strengths, Weaknesses, Opportunities, Threats*: ferramenta utilizada para avaliação de cenários. O SWOT permite identificar forças e fraquezas, que dizem respeito ao

ambiente interno do cenário, e oportunidades e ameaças, que se referem ao ambiente externo ao cenário.

UDDI – *Universal Description, Discovery and Integration*: padrão utilizado para publicar/distribuir *web services*.

UML – *Unified Modeling Language*: ferramenta utilizada para modelar sistemas orientados a objetos.

UP – *Unified Process*: metodologia de desenvolvimento de sistemas. Tem como objetivo transformar os requisitos de um determinado cliente em um sistema de software. É uma metodologia flexível que permite a inclusão ou remoção de atividades, de acordo com as necessidades de cada projeto. É um processo iterativo e incremental, centrado na arquitetura e dirigido por casos de uso.

URI – *Uniform Resource Identifier*: identificador de recursos, é considerado o elemento mais simples e importante do ambiente *web*.

Usabilidade: normalmente está relacionada à interface gráfica da aplicação. No entanto, a usabilidade diz respeito à facilidade de utilização e aprendizado de um sistema.

W3C – *World Wide Web Consortium*: é um consórcio internacional de empresas responsável pelo desenvolvimento e manutenção de diversos padrões relacionados à *web*, incluindo padrões para *web services*, internacionalização, XML, entre outros.

WAR – *Web Application Archive*: é um formato de arquivo, semelhante a um arquivo JAR, porém, é utilizado para distribuir aplicações desenvolvidas para o ambiente *web*. Normalmente um arquivo WAR contém *JavaServer Pages*, *Servlets*, classes *Java*, arquivos XML, etc.

Web 2.0: representa uma nova geração da *web* onde páginas estáticas foram substituídas por conteúdo dinâmico. O ponto central da *Web* 2.0 é a troca de informações entre usuários e os serviços disponíveis na *web*.

WSDL – *Web Services Description Language*: linguagem (baseada em XML) utilizada para descrever *Web Services*. Ou seja, o WSDL descreve as interfaces disponíveis para comunicação com um *web service*.

XML – eXtensible Markup Language: especificação do W3C para geração de linguagens de marcação. O XML pode ser utilizado para codificar documentos, serializar dados, etc.