UNIVERSIDADE DE CAXIAS DO SUL

RODRIGO BOITO

ESTRATÉGIAS ARQUITETURAIS PARA O SISTEMA ERP DO PROJETO DO NUSIS

CAXIAS DO SUL

2013

UNIVERSIDADE DE CAXIAS DO SUL

CENTRO DE COMPUTAÇÃO E TECNOLOGIA DA INFORMAÇÃO
SISTEMAS DE INFORMAÇÃO

RODRIGO BOITO

ESTRATÉGIAS ARQUITETURAIS PARA O SISTEMA ERP DO PROJETO NUSIS

Trabalho de Conclusão de Curso para obtenção do Grau de Bacharel em Sistemas de Informação da Universidade de Caxias do Sul.
Orientadora Prof. Ms Iraci Cristina Silveira de Carli

CAXIAS DO SUL
2013

RODRIGOBOITO

ESTRATÉGIAS ARQUITETURAIS PARA O SISTEMA ERP DO PROJETO NUSIS

Trabalho de Conclusão de Curso para obtenção do Grau de Bacharel em Sistemas de Informação da Universidade de Caxias do Sul.
Orientadora Prof. Ms Iraci Cristina Silveira de Carli

Aprovado em 01/07/2013

Banca Examinadora

_____

Prof. Ms. Iraci Cristina Silveira De Carli

_____

Prof. Dr. Daniel Luis Notari

_____

Prof. Ms. Marcos Eduardo Casa

Dedico este trabalho aos meus pais Sérgio Antônio Boito e Anilse Masiero Boito que não mediram esforços para que eu pudesse chegar até aqui e minha esposa, companheira e amiga Andréia Morello.

AGRADECIMENTOS

Primeiramente gostaria de agradecer a Deus, pela saúde e pela vida que eu tenho, por estar numa família com princípios éticos que me educou de uma maneira em que eu pudesse chegar nesse momento especial. Agradeço a Universidade de Caxias do Sul que além me dar totais condições e me abrir muitas portas, me trouxe conhecimentos para a vida através da convivência que tive com diversas pessoas.

Agradeço aos meus pais e ao meu irmão que sem eles com certeza eu não chegaria até aqui.

Agradeço a minha companheira Andréia Morello que apesar de às vezes não aceitar muito as minhas ausências, principalmente na elaboração deste trabalho, sempre esteve comigo em todos os momentos.

Não posso deixar de agradecer a minha orientadora que me instigou a buscar cada vez mais neste trabalho e soube me orientar de uma forma que eu pudesse chegar ao objetivo desejado.

Por fim agradeço aos meus colegas de trabalho que entenderam minhas ausências e me possibilitaram mais tempo para a realização desta pesquisa.

RESUMO

A partir do aumento da complexidade dos *softwares* tornou-se necessária a definição de arquiteturas de *software*. Decisões tomadas na fase de iniciação do *software*, minimizam o esforço empenhado para uma mudança, do que quando a arquitetura já foi implementada.

Uma questão muito importante nas decisões arquiteturais é atingir os atributos de qualidade desejáveis para o produto tais como, desempenho, manutenibilidade, confiabilidade que não estão diretamente relacionados às funcionalidades do sistema e com papel importante durante o seu desenvolvimento. Implicando na escolha de estruturas arquiteturais, em alternativas de projeto e na forma de implementação.

Avaliar uma arquitetura de *software* em grandes sistemas é algo complexo. É difícil comparar e até entender em um curto espaço de tempo. Além disso, cada interessado tem necessidades diferentes. A aplicação de métodos de análise arquitetural podem diminuir os riscos na definição da arquitetura de *software.* Eles compreendem etapas que auxiliam nas escolhas e cada atributo de qualidade é escrito através de cenários, assim como sua priorização.

Neste trabalho serão apresentadas as estruturas arquiteturais que juntam formam uma arquitetura de software, seus componentes e conectores, padrões de projeto, métodos de análise arquitetural, frameworks de desenvolvimento. A partir destes estudos será definida a arquitetura candidata para o ERP do NUSIS. O software será baseado na Web e a fim de demonstrar que os atributos de qualidade podem ser atendidos pela arquitetura proposta, foi desenvolvido um protótipo de *software* seguindo as características da arquitetura proposta.

Palavras chave: Arquitetura de *Software*, atributos de qualidade, métodos de análise arquitetural, ERP, WEB, ATAM, Spring Framework

ABSTRACT

With the increasing complexity of software has become necessary to define software architectures. Decisions taken in the inception phase of the software, committed effort to minimize the change than when the architecture has been implemented.

A very important issue is the architectural decisions achieve the desirable quality attributes for the product such as performance, maintainability, reliability, which are not directly related to system functionality and important function during development. Implying the choice of architectural structures in design alternatives and the way of implementation.

Evaluate a software architecture for large systems is complex. It is difficult to compare and to understand in a short time. In addition, each stakeholder has different needs. The application of methods of architectural analysis can reduce the risks in the definition of software architecture. They include steps that assist in choices and each quality attribute is written through scenarios, and its prioritization.

This work present the architectural structures that together constitute a software architecture, its components and connectors, design patterns, architectural analysis methods, development frameworks. From these studies will set the candidate architecture for ERP NUSIS. The software is web-based and to demonstrate that quality attributes can be satisfied by the proposed architecture, we developed a prototype software following characteristics of the proposed architecture.

Keywords: *software* architecture, quality attributes, architectural analysis methods, ERP, WEB

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ATAM	Architecture Tradeoff Analysis Method
CBAM	Cost Benefit Analysis Method
CCTI	Centro de Ciências e Tecnologia de Informação
DAO	Data Access Object
DTO	Data Transfer Object
EJB	Enterprise Java Bean
ERP	Enterprise Resource Planning
HTTP	Hyper Text Transfer Protocol
IEC	International Electrotechnical Comission
ISO	International Organization for Standardization
JDBC	Java Database Connectivity
JMS	Java Message Service
JSON	JavaScript Object Notation
MVC	Model-View-Controller
NBR	Norma Brasileira
NUSIS	Núcleo de Sistemas de Informação
OXM	Object Extensive Markup Language
PAC	Presentation-Abstraction-Control
RMI	Remote Method Invocation
RUP	Rational Unified Process
SAAM	Software Architecture Analysis Method
SEI	Software Engineering Institute
SSL	Secure Socket Layer

UML	Unified Modelling Language

SUMÁRIO

1. INTRODUÇÃO

Na década de 90 quando aconteceu a expansão da internet, muitos desenvolvedores passaram a desenvolver WebApps, aplicativos que poderiam ser executados a partir de um navegador, permitindo acesso de qualquer local. Em consequência disto, este tipo de software ganhou cada vez mais importância no mundo corporativo. Em decorrência do aumento da importância, passaram a executar funções, antes somente executadas por *softwares* chamados *desktop.* Deixando de ser um mero sistema de hipertexto para ganhar capacidade computacional. Proporcional a isso os problemas tornaram-se maiores aumentando o grau de complexidade da aplicação. Em virtude do aumento da complexidade dos WebApps se tornou também necessária a definição de uma arquitetura de *software.* (Pressmann, 2006).

Arquitetura de *software* é a organização fundamental do sistema representada por suas estruturas, seus relacionamentos com o ambiente, e pelos princípios que conduzem seu *design* e evolução. Dita o que pode e o que não pode ser feito durante todo o ciclo de vida do software. Além disso, a partir dessas decisões permite a avaliação de uma implementação do sistema a partir de um projeto arquitetural. Por fim a arquitetura de *software* também serve como facilitadora da comunicação entre os vários interessados no sistema, pois ela traduz em suas diversas visões as preocupações de cada membro da equipe (Barbosa, 2009).

O projeto arquitetural é muito importante no processo de desenvolvimento, pois orienta a implementação dos atributos de qualidade do software e auxilia no entendimento de uma solução complexa. Para se chegar à qualidade desejada no *software* é importante o conhecimento de um conjunto de requisitos arquiteturais. Estes requisitos constituem propriedades desejadas ou apresentadas por uma arquitetura de *software*. De posso dos requisitos o arquiteto, busca identificar quais as estruturas arquiteturais são mais adequadas ao sistema que será desenvolvido (Mendes 2002).

A definição dos requisitos arquiteturais segundo Mendes (2002) não é tarefa fácil. A medida que a complexidade de sistemas aumenta estes requisitos arquiteturais passam a ser um aspecto de maior significado quando comparado à escolha de algoritmos ou estruturas de dados.

Para garantir que os requisitos arquiteturais sejam levados em conta é necessário a avaliação dos mesmos através de métodos de análise arquitetural. Desta forma é possível

tentar determinar algumas decisões ou modificações, antes da arquitetura ser construída, ou seja, é importante que a arquitetura seja tratada ainda na fase de concepção. (Mendes, 2002).

1.1 PROBLEMA DE PESQUISA

A Universidade de Caxias do Sul possui um Núcleo de Avaliação, Seleção, Desenvolvimento e Aplicação de Sistemas de Informação (NUSIS), no qual existe um projeto com o intuito de disponibilizar sistemas de informação para empresas de pequeno porte. O objetivo é criar um ambiente para alunos desenvolverem sistemas podendo aplicar em situações reais. O desenvolvimento produzirá sistemas de diversos tipos e, a partir de sua como: Sistemas Estratégicos, Base de dados, Portais Corporativos, ERP (Enterprise Resource Planning). Esse último fará parte do objeto de estudo.

Os requisitos funcionais ainda não foram elicitados, o que se tem conhecimento é que o sistema será modularizado por área funcional. Os atributos de qualidade serão definidos e especificados ao longo deste trabalho.

O ERP será uma aplicação baseada na Web, consolidando o conteúdo desenvolvido nas disciplinas dos cursos do CCTI (Centro de Computação e Tecnologia da Informação). Utilizando o paradigma de orientação de objetos e como linguagem Java.

O processo de desenvolvimento será o Processo Unificado Ágil, iterativo, incremental e evolutivo. É centrado na arquitetura e orientado por casos de uso e utiliza como técnica de *model*agem a UML (Unified *Model*ing Language).

O *software* será desenvolvido por equipes diferentes em momentos diferentes, sempre formadas por alunos em estágio ou desenvolvendo trabalhos de conclusão. Esta forma caracteriza uma alta rotatividade da equipe, além da pouca experiência da mesma em desenvolvimento de sistemas.

Uma das atividades para o desenvolvimento do ERP do Projeto do NUSIS são as estratégias para definição de arquiteturas de *software* na fase de concepção. Essas estratégias darão subsídios para o desenvolvimento do sistema com os atributos e a qualidade exigidos. Os critérios para garantir a qualidade do *software* serão definidos ao longo do trabalho.

1.2 QUESTÕES DE PESQUISA

Baseado no problema de pesquisa descrito anteriormente foi criada a seguinte questão de pesquisa:

Quais são as estratégias, em relação à arquitetura de *software*, mais apropriadas ao projeto do ERP do Nusis na fase de concepção?

1.3 OBJETIVOS

O objetivo deste trabalho é propor estratégias para arquitetura de *software* a ser utilizada no sistema de ERP do projeto do NUSIS. Entre as estratégias está a definição de uma arquitetura candidata, bem como a definição dos atributos de qualidade desejados para o ERP.

A avaliação dessa arquitetura se dará através da aplicação de um método de Análise Arquitetural.

Essas estratégias serão desenvolvidas na fase de concepção e direcionarão o desenvolvimento de iterações, inclusive no levantamento de requisitos funcionais que poderão surgir durante o projeto, apesar de já serem conhecidos alguns requisitos do sistema.

A fim de demonstrar que a arquitetura candidata pode atender aos atributos de qualidade desejados um protótipo de software deverá ser desenvolvido a partir da arquitetura candidata gerada.

1.4 METODOLOGIA E ORGANIZAÇÃO DO TRABALHO

Para a realização deste trabalho a metodologia seguirá os seguintes passos:

Primeira etapa: Levantamento do material bibliográfico sobre Arquiteturas de *Software*. Realizando comparações a fim de trazer as melhores estratégias para a definição da arquitetura candidata do ERP do NUSIS.

Segunda Etapa: Elicitação dos atributos de qualidade desejados para o sistema a fim de levantar as estratégias da arquitetura candidata que será proposta para o ERP do NUSIS.

Analisando outros sistemas com características similares e em levantamento com os coordenadores do projeto. Buscando bibliografia que possa abalizar o que é importante para um sistema desse tipo.

Terceira Etapa: Analisar as estratégias arquiteturais levantadas para a utilização no ERP do NUSIS. Confrontar com os atributos que serão necessários para o ERP e que serão definidos durante o projeto.

Quarta etapa: Análise da arquitetura candidata a partir da utilização de um Método de Análise Arquitetural.

Quinta etapa: Desenvolver um protótipo de *software* utilizando a arquitetura candidata proposta. O protótipo deverá ser desenvolvido na linguagem Java utilizando o processo unificado centrado na arquitetura e deverá abordar um caso de uso ou mais, o suficiente para atender aos atributos de qualidade desejados.

O trabalho está organizado em quatro capítulos, onde no capítulo 1 é apresentada a introdução compreendendo a contextualização do assunto, problema de pesquisa, questão de pesquisa, objetivos e metodologia do trabalho.

No capítulo 2, são apresentados estudos bibliográficos realizados sobre qualidade de *software*, arquitetura de *software* composta por estruturas arquiteturais, padrões de projeto, arquitetura de interface com usuários, requisitos arquiteturais, métodos de análise arquitetural e documentação da arquitetura.

No capítulo 3, baseado no método de análise arquitetural, são apresentados os atributos de qualidade para o projeto do ERP do Nusis, bem como decisões como a utilização de um framework de desenvolvimento.

No capítulo 4, será apresentado o protótipo de software que visa garantir que os atributos de qualidade sejam atendidos.

2. ARQUITETURA DE *SOFTWARE*

Não há uma definição universal para arquitetura de *software* e existem muitos termos de diversos autores. O SEI (*Software* Engineering Institute) apresenta inúmeras definições para esse assunto, a definição a seguir é coerente com o que será apresentado ao longo da pesquisa.

"A arquitetura de *software* de um programa ou sistema computacional é a estrutura ou estruturas do sistema, que envolvem elementos de *software*, propriedades externamente visíveis desses elementos e o relacionamento entre eles" (Bass, 2006 p.21).

Para Larman (2007) arquitetura de *software* tem a ver com ideias grandes e complexas, utilizando padrões, responsabilidades e conexões de um sistema ou sistema de sistemas (Larman, 2007).

Não é possível dizer se uma arquitetura de *software* é boa ou ruim, sem analisar a qual propósito ela se insere. Arquiteturas são mais ou menos aptas para determinados projetos (Bass, 2006). Segundo Bass (2006) existem algumas recomendações que devem ser seguidas para a definição de uma boa arquitetura, assim como a definição dos atributos funcionais e a priorização dos atributos de qualidade de *software*, dando subsídios para a escolha de estruturas arquiteturais.

Em conjunto com as estruturas arquiteturais está à exigência da qualidade do produto final, dessa forma o projetista deve decidir a arquitetura na camada de interface que mais se adequa aos atributos de qualidade desejados para o *software* a ser desenvolvido (Mendes, 2002).

Um processo de desenvolvimento centrado na arquitetura considera a arquitetura de *software* como fator preponderante no processo. Esse processo pode iniciar com o arquiteto de posse de um conjunto de requisitos arquiteturais, buscando identificar quais estruturas arquiteturais, padrões arquiteturais, padrões de projeto, arquitetura de interface, as utilizando para satisfazer aos atributos de qualidade desejados (Mendes, 2002).

Uma maneira de aplicar as estruturas e padrões é com a adoção de um framework de desenvolvimento. O *framework* já possui várias decisões e padrões de projeto embutidos nele, além de elevar o nível de qualidade da aplicação, um alto grau de reutilização de código e aumento na produtividade (Barreto, 2006).

Como forma de auxiliar o desenvolvimento da arquitetura, estão os métodos de análise arquitetural, os quais dão subsídios para que as escolhas se tornem menos arriscadas e

minimizando o esforço de mudanças, já que a arquitetura ainda não foi implementada (Mendes, 2002).

Para Bass (2006) a arquitetura de *software* desempenha um papel preponderante no desenvolvimento de sistema e na organização que o produz. Ela serve como um modelo para o desenvolvimento de um sistema definindo as atribuições de trabalho que devem ser realizadas pelas equipes de projeto e implementação, e é o principal meio para se chegar aos atributos de qualidade, porém até mesmo a arquitetura mais perfeita se tornaria inútil se os interessados não a entenderem. Faz-se necessária então a geração de uma documentação com detalhes suficientes que tornem essa arquitetura entendível a todos os interessados.

2.1 ARQUITETURA DE *SOFTWARE* NO PROCESSO DE DESENVOLVIMENTO DE SOFTWARE

A arquitetura de *software* é trabalhada nas diversas fases do Processo Unificado: iniciação, elaboração, construção e transição. Cada uma dessas fases pode conter uma ou mais iterações (Booch, 2000).

Entre os objetivos da fase de iniciação está a definição do escopo do projeto (requisitos e as restrições mais importantes) e a síntese de uma possível arquitetura (estimando os custos, programação e recursos) (Scott, 2003). O foco desta pesquisa é a fase de concepção na disciplina de análise e design, que tem como objetivo realizar uma síntese arquitetural. A finalidade da síntese arquitetural é detalhar o fluxo de trabalho construindo e avaliando um conceito de arquitetura candidata mostrando o que existe para satisfazer seus principais requisitos (Scott, 2003).

Na fase de elaboração a meta é estabelecer uma linha de base para a arquitetura, a fim de fornecer uma base para o esforço na fase de construção. Entre os objetivos desta fase estão diminuir e tratar os riscos, estabelecer a arquitetura base derivada do tratamento dos cenários significativos do ponto de vista da arquitetura (Booch, 2000).

A fase de construção deve esclarecer os requisitos restantes e concluir o desenvolvimento a partir de uma arquitetura base. Como objetivos tem, minimizar os custos de desenvolvimento, atingir a qualidade adequada, concluir a análise, design, desenvolvimento e testes de todas as funcionalidades necessárias (Booch, 2000).

Para se chegar a uma arquitetura candidata na fase de concepção, é necessário tomar algumas decisões. Para facilitar a tomada de decisões convém a utilização de padrões arquiteturais, bem como padrões de projeto e existem inúmeros. Para Gamma (2000) padrões de projeto tornam mais fácil reutilizar projetos e arquiteturas bem-sucedidas. Ajudam a escolher alternativas que tornam um sistema reutilizável e a evitar alternativas que comprometam a reutilização. Padrões arquiteturais expressam o esquema ou organização estrutural fundamental de sistemas de *software* ou hardware (Masiero, 2001).

2.2 ARQUITETURA DE *SOFTWARE* E QUALIDADE

Segundo Sommerville (2011) a arquitetura de *software* afeta diretamente aos atributos de qualidade do sistema. A norma ISO/IEC 9126 que trata da qualidade do produto de *software* pode apoiar o processo de definição destes atributos2. Ela está dividida em quatro partes:

I. Modelo de Qualidade
II. Métricas Externas
III. Métricas Internas
IV. Métricas Qualidade em Uso

Para a definição de um esboço de arquitetura será utilizada a primeira parte, Modelos de Qualidade. Ela lista os atributos de qualidade do *software* e é composta de duas subpartes: qualidade interna e externa e qualidade em uso.

A primeira parte trata especificamente de seis categorias as quais são dividas em subcategorias resultantes dos atributos internos do *software*. A segunda parte especifica quatro características de qualidade em uso, sendo o efeito combinado das seis categorias especificadas na primeira parte. Essas características podem ser aplicáveis para todos os tipos de *software* como sistemas de computadores e firmwares e podem trazer uma terminologia consistente para tratar de qualidade do produto de *software* (NBR ISO/IEC 9126, 2003).

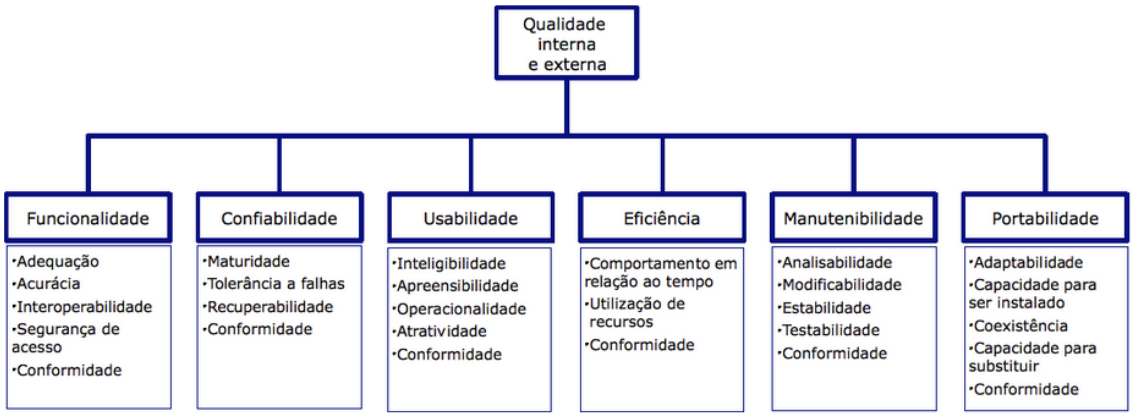
2.2.1 Qualidade Interna e Externa

Os requisitos de qualidade externa são analisados derivados das necessidades de qualidade dos usuários. Esses requisitos são utilizados como metas para a validação de vários estágios de desenvolvimento. A qualidade externa é vista quando o *software* estiver sendo executado e pode ser analisada em um ambiente simulado onde a maioria dos erros podem ser descobertos e corrigidos (NBR ISO/IEC 9126, 2003).

Os requisitos de qualidade internos especificam o nível de qualidade sob o ponto de vista interno do produto. São utilizados para especificar as propriedades dos produtos intermediários incluindo modelos estáticos e dinâmicos, outros documentos e código-fonte. Podem ser utilizados para definir estratégias de desenvolvimento e critérios de avaliação durante o desenvolvimento (NBR ISO/IEC 9126, 2003).

A norma 9126 define um modelo de qualidade de *software* onde podem ser definidas as metas de qualidade do produto de *software* final e os intermediários. São categorizadas seis características (funcionalidade, confiabilidade, usabilidade, eficiência, manutenibilidade e portabilidade) onde cada uma delas é dividida em mais sub-características podendo ser medidas por métricas internas e externas, como ilustra a Figura 1 (NBR ISO/IEC 9126, 2003).

Figura 1 – Modelo de Qualidade para Qualidade Externa e Interna



Fonte: ISO/IEC 9126-1, 2003.

O atributo funcionalidade é a capacidade do produto em prover soluções que atendam às necessidades implícitas e explicitas, quando o *software* estiver sendo executado sob condições especificas.

Confiabilidade de um *software* é a capacidade que ele tem de executar as funções requisitadas, mantendo um nível de desempenho adequado e nas condições especificadas.

A usabilidade é a característica que trata diretamente como o usuário vai se relacionar com o *software* especificado. Essa característica está focada na capacidade de compreensão, aprendizado, operação e atratividade ao usuário.

Eficiência é a capacidade do produto de *software* apresentar desempenho apropriado, relativo à quantidade de recursos usados, sob as condições especificadas.

Manutenibilidade é a capacidade do *software* de ser modificado, podendo incluir correções, novas funcionalidades, mudança nos requisitos, melhorias ou adaptações por mudanças no ambiente.

A portabilidade trata da capacidade do *software* de ser transferido de um ambiente para outro, podendo ser um ambiente organizacional, de hardware ou de *software*. (NBR ISO/IEC 9126, 2003).

2.2.2 Qualidade em Uso

A qualidade em uso é a visão da qualidade na perspectiva do usuário. Ela trata da capacidade do *software* de permitir que os usuários atinjam as metas especificadas como eficácia, produtividade, segurança e satisfação. Os atributos de qualidade em uso são ilustrados pela Figura 2 (NBR ISO/IEC 9126, 2003).

Figura 2 – Qualidade em Uso NBR ISO/IEC 9126 Parte 1

Fonte: ISO/IEC 9126-1, 2003

A seguir serão definidos os atributos de qualidade em uso.

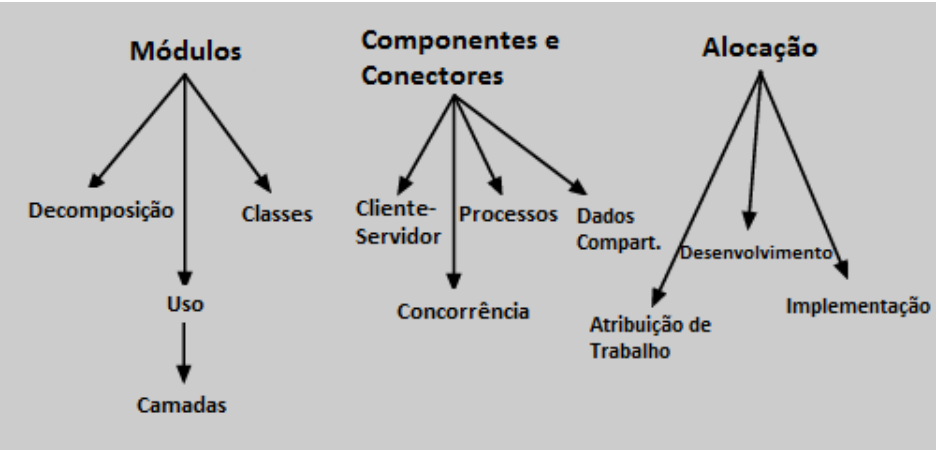
a) Eficácia – Capacidade do usuário de atingir as metas especificadas com acurácia e completitude no contexto especificado
b) Produtividade – É capacidade do *software* de permitir que o usuário utilize os recursos apropriados em relação à eficácia obtida. Nessa característica os recursos podem ser tempos para concluir uma tarefa, esforço do usuário, materiais ou custos financeiros.
c) Segurança – Capacidade do *software* de apresentar a segurança adequada em relação a riscos de danos às pessoas, negócios, *software* ou propriedades ou ambiente.
d) Satisfação – Capacidade do produto de satisfazer ao usuário. Nesse caso são as respostas que o usuário dá em relação à interação com o produto (NBR ISO/IEC 9126, 2003).

2.3 ESTRUTURAS ARQUITETURAIS

Propor uma arquitetura candidata adequada aos atributos de qualidade do *software* exige a tomada de muitas decisões sobre as estruturas arquiteturais. Uma estrutura arquitetural permite que um profissional determine características dos subsistemas e conectores do sistema, topologia da arquitetura, restrições e mecanismos de interação entre os elementos (Bass, 2006).

A seguir serão apresentadas as estruturas arquiteturais. Bass (2006) as divide em três grupos: estrutura de módulos, componentes e conectores, e alocação. Conforme mostra Figura 3, estes três grupos são divididos em sub-características (Bass, 2006).

Figura 3 – Estruturas Arquiteturais



Fonte: Bass, 2006

Para Bass (2006), estes são três grandes tipos de decisão que envolvem um projeto arquitetural e podem responder as seguintes perguntas:

- O sistema será estruturado como um conjunto de unidade de código (Módulos)?
- O sistema será estruturado como um conjunto de elementos em tempo de execução (Componentes) e suas interações (Conectores)?
- Como o sistema irá se relacionar com as estruturas "não *software*" em seu ambiente (CPU´s, sistema de arquivos, redes, equipes de desenvolvimento)?

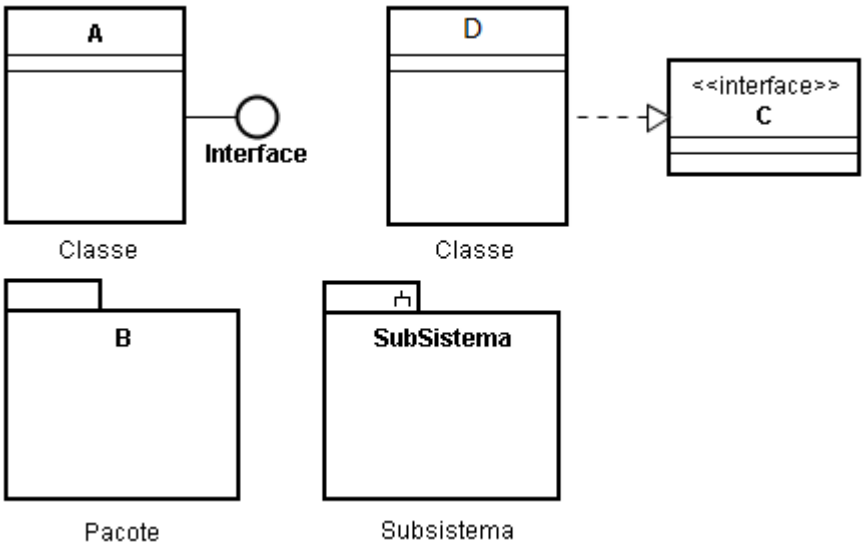
2.3.1 Módulos

Quando os arquitetos se deparam com sistemas de grande porte e complexos, geralmente eles dividem este sistema em partes menores. Estas partes são denominadas módulos. Compreender a modularidade de um sistema é entender cada parte separadamente, tornando mais fácil a manutenibilidade. A modularidade ajuda a isolar a fonte do problema a um único componente. Um sistema modular implica em módulos com uma alta coesão e baixo nível de acoplamento. Isso permite considerar os módulos como caixas-pretas possibilitando lidar com cada funcionalidade do sistema de forma independente (Mendes, 2002).

Na estrutura em módulos é dada menos ênfase em relação ao funcionamento do *software* em tempo de execução. Estruturas modulares nos permitem responder a alguns questionamentos como: Qual é a principal funcionalidade de cada módulo? Quais módulos vão se comunicar com outros por relacionamento de generalização ou especialização? (Bass, 2006).

A UML fornece uma variedade de formas para se representar diferentes tipos de módulos. A Figura 4 mostra alguns exemplos. A classe A e D são especializações orientadas a objetos de um módulo. O Pacote B que pode ser utilizado onde o agrupamento de funcionalidades é necessário, como na representação de camadas e classes. O subsistema que pode ser usado se uma especificação de interface é requerida.

Figura 4 – Exemplos de Módulos em notações UML



Fonte: Bass, 2006

Bass (2006) divide a estrutura modular em estrutura de decomposição, estruturas de uso, estruturas de camadas e estruturas de classes (Bass, 2006).

2.3.1.1 Estrutura de Decomposição

Quando os sistemas a serem desenvolvidos são complexos e exigem que uma grande equipe de desenvolvimento esteja alocada para tal, deve-se pensar em decomposição. Os módulos maiores do sistema são divididos em submódulos e assim recursivamente até que fiquem suficientemente pequenos para serem facilmente compreendidos. O Arquiteto enumera quais serão as funções desempenhadas por cada unidade do *software* e muitas vezes são usadas como base para a organização do projeto de desenvolvimento incluindo a estrutura de documentação, integração e planos de teste (Bass, 2006)

Segundo Bass (2006), um módulo pode definir um grupo de procedimentos, alguns públicos outros privados e ainda um conjunto privado de estrutura de dados. Dessa forma a interface do módulo revela somente o que é suscetível a mudanças, os detalhes ocultos das interfaces dos módulos são secretos, esse é o princípio do ocultamento da informação. As interfaces públicas podem ser acionadas de qualquer outro módulo, por um conjunto de procedimentos de acesso e permitem que um módulo interaja com uma informação encapsulada de outro módulo. Abaixo são listadas as metas especificas para a decomposição:

- Cada módulo deve ser simples o bastante para que possa ser plenamente entendido
- Deve ser possível modificar um módulo sem ter conhecimento da implementação dos outros módulos e sem afetá-los (Bass, 2006).

O uso da técnica de ocultamento de informação traz maiores benefícios quando são necessárias modificações na fase de testes e após quando for necessária manutenção. Como a maior parte dos dados e procedimentos estão ocultas de outras partes do *software*, erros oriundos das modificações são menos prováveis de serem propagados para outros locais do *software* (Pressman, 2006).

2.3.1.2 Estrutura de Usos

As unidades dessa estrutura também são os módulos, procedimentos ou recursos das interfaces dos módulos. A estrutura de uso pode ser utilizada por engenheiros de sistemas que facilmente podem estender e adicionar funcionalidades ou a partir das quais subconjuntos funcionais podem ser extraídos. Isso possibilita um desenvolvimento incremental.

A estrutura de decomposição não fornece nenhuma informação sobre a interação no *software* em tempo de execução. Para esses casos a responsável por essas interações é a estrutura de uso, preocupando-se com as relações entre os módulos através dos procedimentos que são invocados por eles. A estrutura de usos tem a responsabilidade de ditar quais os procedimentos estão autorizados ou não a utilizar outros procedimentos (Bass, 2006).

2.3.1.3 Camadas

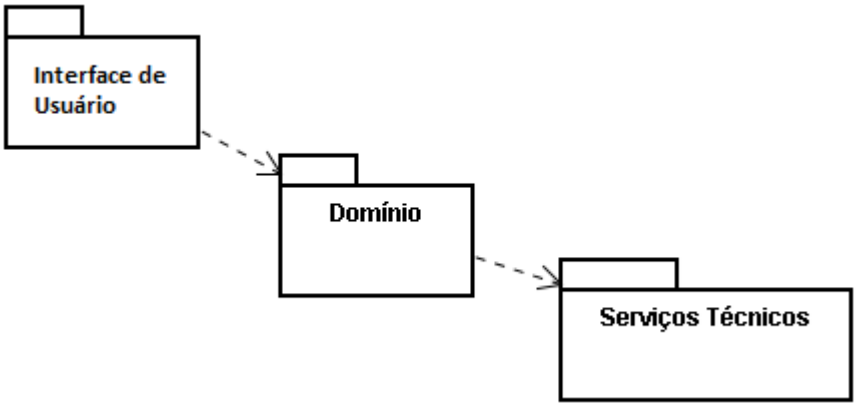
Quando as relações da estrutura de uso são cuidadosamente controladas, surge um sistema de camadas que nada mais é que um conjunto coerente de funcionalidades relacionadas (Bass, 2003).

O conceito camadas se tornou mais visível a partir de década de 90 com o surgimento dos sistemas cliente-servidores. Os sistemas funcionavam em duas camadas onde a camada cliente era responsável por mostrar a interface com o usuário e algum código da aplicação e a camada servidor era normalmente um banco de dados relacional.

Uma camada é um elemento de uma grande parte do sistema como classes, pacotes e subsistemas e com responsabilidade coesiva. Essas camadas trabalham de tal forma que as mais altas solicitam serviços para as mais baixas (Larman, 2007).

Segundo Bass (2003), em uma estrutura estritamente em camadas, a camada n só pode utilizar serviços da camada n-1. As camadas geralmente são pensadas como abstrações escondendo detalhes de implementação abaixo das camadas superiores, gerando portabilidade, a Figura 5 ilustra uma estrutura em camadas. Para Larman (2007) em sistemas de informação as camadas mais altas podem solicitar informação de várias camadas mais baixas, nesse caso a camada de IU poderia solicitar um serviço diretamente para as camadas de serviços técnicos, essa forma é denominada uma estrutura em camadas relaxada. Cada camada devem ter interesses semelhantes e são geralmente dividas em três camadas principais, interface com usuário, regras de negócio e persistência.

Figura 5 – Diagrama de Pacotes da arquitetura Lógica em camadas



Fonte: Larman, 2007

- Interface com Usuário (IU) – como o nome já diz essa camada oferece condições para que o usuário execute as tarefas no sistema (Larman, 2007). Essa camada tem a responsabilidade de traduzir ações do usuário em comandos que possam interagir com as camadas de domínio e a camada de fonte de dados (Fowler, 2006).
- Regras de Negócio – Essa camada contém objetos de *software* que representam os conceitos de domínio que satisfazem os requisitos da aplicação (Larman, 2007). Envolve cálculos baseados na entrada de dados e nos dados armazenados, validação de qualquer dado proveniente da camada de aplicação e com a compreensão exata de qual lógica executar dependendo dos dados recebidos da camada de apresentação (Fowler, 2007).

- Persistência (Fonte de Dados) – são objetos com propósitos gerais oferecendo serviços técnicos de apoio assim como conexão com o banco de dados e registro de erros. Essa camada geralmente independe da aplicação e podem ser reusáveis em outros sistemas (Larman, 2007). Para Fowler, esta camada na maioria das aplicações é um banco de dados responsável, antes de tudo, pelo armazenamento de dados persistentes (Fowler, 2007).

Um projeto em camadas tem o objetivo de organizar a estrutura lógica de forma coesa e com separação de interesses. Dessa forma cada camada tem responsabilidades distintas sendo as camadas inferiores de baixo nível e as superiores mais específicas da aplicação. O acoplamento ocorre das camadas superiores para as inferiores o que possibilita a reutilização das camadas inferiores em outras aplicações como, por exemplo, a utilização em plataforma web (Larman, 2007).

2.3.1.4 Estrutura de Classe ou generalização

As unidades modulares nesta estrutura são chamadas de classes. Este ponto de vista se baseia no conceito de herança onde objetos que tem comportamento e capacidades semelhantes podem herdar atributos de outros objetos. Essas são denominadas subclasses. A estrutura de classes permite o reuso e adição incremental de novas funcionalidades (Bass, 2006).

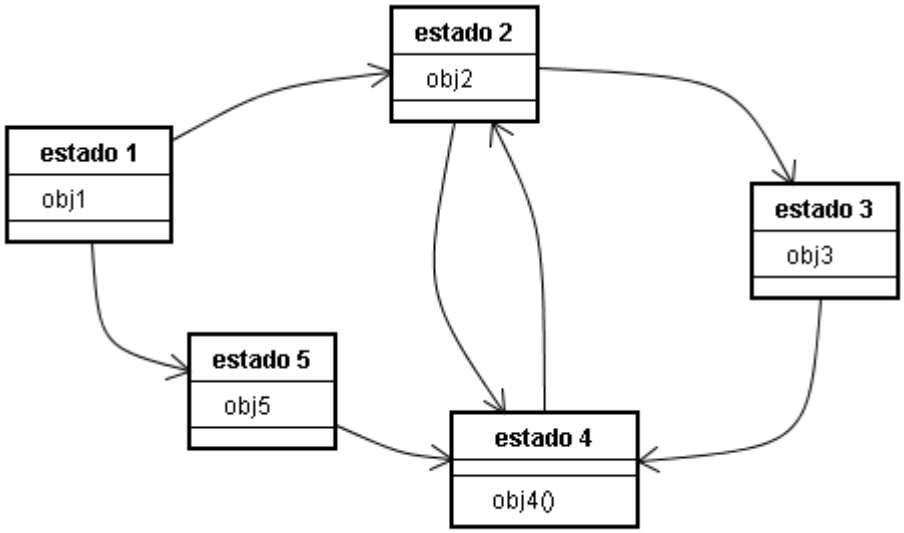
Para Sommerville (2006), a abordagem orientada a objetos é comum hoje em dia, principalmente para desenvolvimento de sistemas interativos. Isto significa desenvolver o sistema pensando em um *model*o de objetos, utilizando objetos e programando em linguagem orientada a objetos, como Java ou C++. “A decomposição orientada a objetos diz respeito às classes de objetos, seus atributos e suas operações” (Sommerville, 2006).

Mendes faz uma analogia considerando uma empresa que possui diversos departamentos como, por exemplo, vendas, contabilidade, onde cada departamento possui seu pessoal e tarefas associadas. Cada departamento possui seus próprios dados e cada pessoa que faz parte dele atua sobre os seus dados. Dessa forma, ao dividir a empresa em departamento se torna mais fácil compreender suas necessidades e melhora o seu controle. A abordagem orientada a objetos permite a organização de programas ao mesmo tempo em que ajuda a

manter a integridade dos dados do sistema. Em um projeto orientado a objetos, os projetistas podem encapsular dados, dessa forma teríamos a ocultação de informações (Mendes, 2002).

A estrutura arquitetural de classes enxerga o sistema como um conjunto de objetos que se comunicam através de canais de comunicação. Nesse caso quando um objeto necessita se comunicar com outro, este deve conhecer a identidade do objeto ao qual enviará a mensagem, conforme ilustra a Figura 6, onde existe uma troca de informação entre os objetos (Mendes, 2002).

Figura 6 – Arquitetura de um Projeto Orientado a Objetos



Fonte: Mendes, 2002

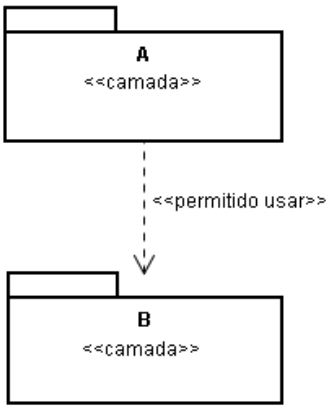
Uma desvantagem em ter a obrigatoriedade de conhecer a identidade dos objetos para se comunicar pode ser em relação ao reuso, pois em cada mudança deve-se alterar a especificação das classes. Além disso, se a manutenibilidade for um atributo de qualidade considerado importante deve-se pensar que qualquer manutenção necessária deve afetar um número reduzido de objetos. Ainda no quesito atributos de qualidade se for levado em conta o desempenho do sistema em cada cenário de uso mais frequente deveria ocorrer em um número menor de objetos. Minimizando a comutação de contexto, o que poderia vir a comprometer o desempenho do sistema (Mendes, 2002).

2.3.2 Componentes e Conectores

Nessa estrutura os elementos são componentes em tempo de execução e os conectores são o veículo de comunicação entre esses componentes. A estrutura componentes e conectores permite responder as seguintes questões: Quais são os principais componentes de execução e como eles interagem? Quais partes do sistema podem rodar em paralelo? (Bass, 2006)

Segundo Bass (2006) existe um grande número de alternativas para documentar esta estrutura em UML. Para ele uma candidata para representar a estrutura componentes e conectores é demonstrada na Figura 7.

Figura 7 – Representação de camadas em UML



Fonte: Bass, 2006

Bass (2006) divide esta estrutura em diferentes padrões:

a) Cliente-servidor onde os componentes são os clientes e servidores e os conectores são os protocolos e mensagens que eles compartilham para realizar os trabalhos do sistema.
b) Concorrência que permite ao arquiteto determinar onde o paralelismo pode ocorrer, nesse caso as unidades são threads lógicas. As threads lógicas podem ser executadas separadas das threads físicas. É utilizada no início do projeto para identificar os requisitos as questões que envolvem execução simultânea.
c) Processos que permitem aos engenheiros de *software* darem desempenho e alta disponibilidade para o *software* através de visão física de como o *software* está implementado.

d) Dados Compartilhados ou Repositório compreende a criação, armazenamento e acesso aos dados. Mostra como os dados são produzidos e consumidos por elementos de *software* em tempo de execução e são utilizadas para assegurar um bom desempenho e integridade dos dados.

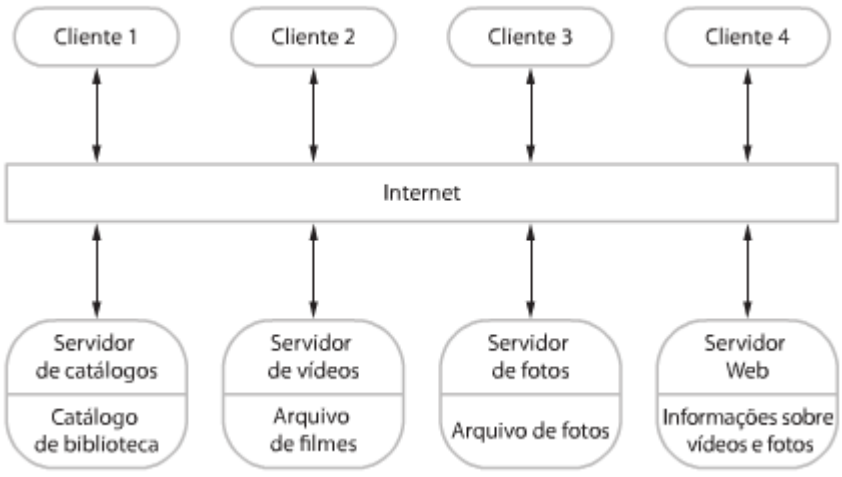
2.3.2.1 Estrutura Cliente-Servidor

A estrutura cliente-servidor é uma boa prática quando o sistema é composto por um grupo de cliente e servidores. Basicamente consiste na separação de interesses (manutenibilidade) para uma distribuição física e balanceamento de carga (desempenho em tempo de execução) (Bass, 2006).

A utilização do termo Cliente-Servidor não necessariamente se refere a um mapeamento 1:1, em geral se refere processos lógicos e não a computadores físicos nos quais esses processo são executados (Sommerville, 2006)

Coulouris (2007) cita também que servidores podem ser clientes de outros servidores, servidores web e a maioria dos outros serviços de internet são clientes do serviço DNS (Domain Name System). A Figura 8 demonstra um sistema multiusuário baseado na internet para fornecimento de uma biblioteca de filmes e fotos.

Figura 8 – Sistema Cliente-Servidor



Fonte: Sommerville, 2011.

Segundo Mendes (2002) a tecnologia cliente-servidor é tradicionalmente implementada de duas formas:

a) Arquitetura Cliente-Servidor de duas camadas (*two-tier-client-server*) – esse tipo de sistema exige uma interface cliente para executar a aplicação e um servidor de banco de dados para gerenciar as transações. É fácil de ser gerenciado, mas quando o volume de transações é baixo, além de exigir que o cliente execute uma grande quantidade de funcionalidades tornando-o grande e complexo.
b) Arquitetura Cliente-Servidor de Múltiplas camadas (*multi-tier-client-server*) – Nessa arquitetura o problema que acontece na arquitetura cliente-servidor de duas camadas foi resolvido, com ela a parte da lógica específica da aplicação é movida para uma camada central. Essa camada é composta de um servidor de aplicação que é responsável pela mediação entre clientes e recursos. Com isso os projetistas podem buscar um melhor desempenho da aplicação e a diminuição das funcionalidades que antes eram executadas pelos clientes (Mendes, 2002).

2.3.2.2 Concorrência

Bass (2006) define concorrência por processamento de diferentes fluxos de eventos em diferentes threads ou pela criação de threads adicionais para processar um conjunto diferente de atividades.

Para Fowler (2006) a concorrência é um dos aspectos mais problemáticos no desenvolvimento de *software*, sempre que se têm múltiplos processos ou threads manipulando os mesmos dados, pode-se deparar com problemas de concorrência. Para ele é muito difícil levantar todos os possíveis cenários que podem ocasionar erros e, além disso, é difícil de testar. (Fowler, 2006).

Segundo Bass (2006), na visão de concorrência podem ser *model*adas atividades paralelas e sincronização. Esta *model*agem vem a ajudar na identificação de problemas como *deadlocks* (ocorrem quando duas ou mais tarefas bloqueiam uma à outra permanentemente), questões de consistência de dados, entre outros. Nesta estrutura os componentes são instancias de módulos e os conectores são as *threads* virtuais (descreve a execução do sistema ou parte dela), que não devem ser confundidas com *threads* de sistema ou processos que implicam em outras propriedades como alocação de memória e processadores (Bass, 2006).

Fowler (2006) cita duas soluções importantes para problemas de concorrência: isolamento e imutabilidade. Por meio de isolamento somente um agente ativo pode alterar o dado, isto é, se o usuário “a” estiver editando um arquivo o usuário “b” pode visualizar, mas não podem alterá-lo. Para Fowler (2006), só se tem problema de concorrência quando dados que estiverem sendo compartilhados puderem ser modificados, uma maneira de evitar conflitos é não permitir que os dados imutáveis sejam alterados. Para ele a identificação de dados como imutáveis ou quase todo o tempo como imutáveis, pode-se relaxar a preocupação com a concorrência.

Quando existem dados mutáveis que não se pode isolar, Fowler (2006) cita duas formas: bloqueio otimista e pessimista.

a) Bloqueio Otimista – caso dois usuários queiram editar um arquivo, ambos fazem uma cópia do arquivo e podem editar livremente. O primeiro que terminar, grava o seu trabalho, já o segundo quando tenta gravar sua confirmação é rejeitada, pois o sistema percebe um conflito.
b) Bloqueio Pessimista – no caso do pessimista o primeiro que edita o arquivo já inviabiliza que o segundo possa editar, até que o primeiro finalize suas modificações (Fowler, 2006).

2.3.2.3 Processos

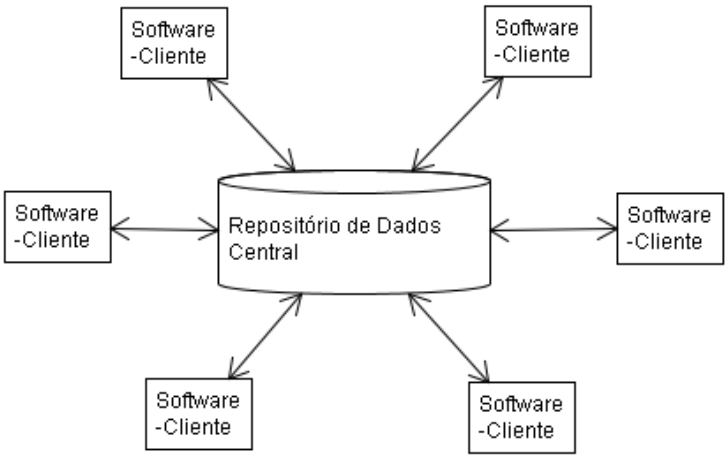
Essa unidade se preocupa com os aspectos dinâmicos da execução do sistema. As unidades são processos ou *threads* físicas, que são conectadas umas as outras por comunicação, sincronização e/ou operações de exclusão. Esta relação mostra como os componentes e conectores são anexados (Bass, 2006). A Estrutura de Processo não é objeto desta pesquisa.

2.3.2.4 Dados Compartilhados ou Repositório

Para Mendes (2006) uma estrutura de dados compartilhados ou repositório possui dois componentes básicos, um conjunto de dados compartilhados e um conjunto de clientes independentes que tem acesso e atualizam este repositório (Mendes, 2006).

Segundo Pressman (2006) nesta estrutura o repositório de dados fica no centro e dá suporte a outros componentes que atualizam, adicionam, retiram ou modifica os dados contidos no repositório, um exemplo típico é ilustrado na Figura 9.

Figura 9 – Arquitetura de Repositório



Fonte: Pressman, 2006

Mendes (2002) e Pressman (2006) citam duas variantes em relação à arquitetura de repositório, passivo e ativo. No repositório passivo o *software*-cliente tem acesso aos dados independentemente de quaisquer mudanças nos dados ou nas ações dos outros clientes. (Pressman, 2006). Já no repositório ativo que também pode ser chamado de Quadro Negro a base de dados compartilhada notifica os clientes que estão conectados, a respeito das mudanças ocorridas e que possam interessar a eles (Mendes, 2002).

Como mostrou a Figura 9, a topologia da arquitetura de repositório é tipicamente em estrela e tem como principal característica o suporte que oferece a qualidade de integração (Mendes, 2002).

2.3.3 Alocação

A estrutura de alocação estabelece um relacionamento entre os elementos de *software* e um ou mais elementos de ambientes externos, onde o *software* é criado ou executado. Ela responde as seguintes questões: Em quais arquivos cada elemento deve ser armazenado durante o desenvolvimento, testes e construção do sistema? Qual é a atribuição de elementos

de *software* para as equipes de desenvolvimento? Em qual processador cada elemento de *software* deve executar? (Bass, 2006).

Bass (2006) divide a estrutura de alocação em: Desenvolvimento, Implantação e Atribuição de Trabalho.

2.3.3.1 Estrutura Desenvolvimento

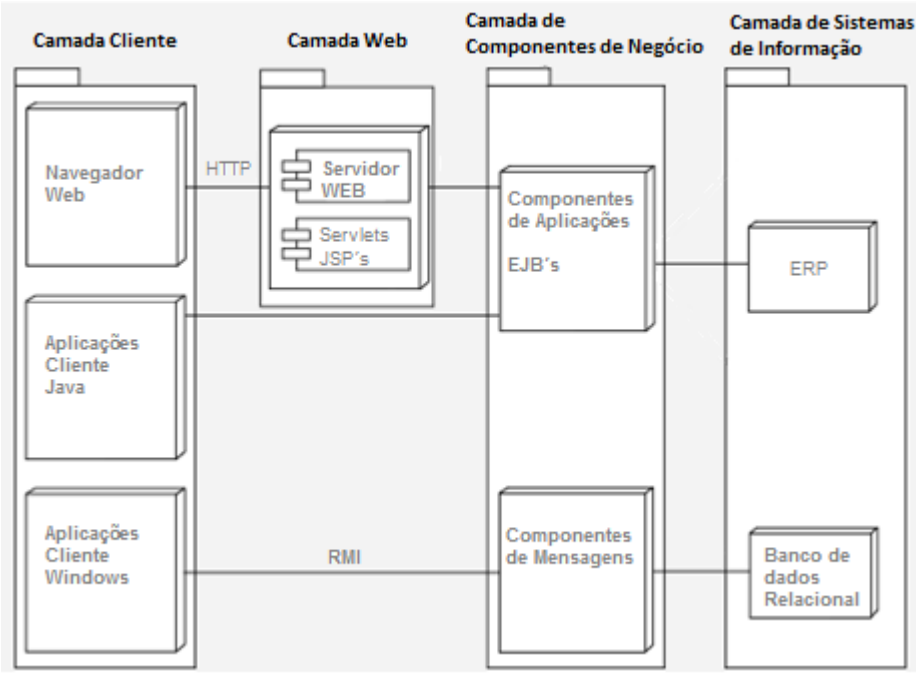
A estrutura de desenvolvimento mostra como o *software* é atribuído no hardware disponível e os elementos de comunicação. Como vão residir os elementos de *software* fisicamente. Essa estrutura permite ao engenheiro pensar em desempenho, integridade de dados, disponibilidade e segurança. Particularmente em sistemas distribuídos ou paralelos (Bass, 2006). Para Kruchten (2000) a estrutura de desenvolvimento mostra como vários executáveis e outros componentes em tempo de execução são mapeados para plataformas adjacentes ou nós de computação abordando questões de instalação, implantação e desempenho.

A estrutura de desenvolvimento ajuda a determinar a alocação de componentes de *software* no hardware disponível, pode se utilizar táticas como replicação, oferendo auxilio na obtenção de alto desempenho ou confiabilidade com a implantação de replicas em diferentes processadores (Bass, 2006).

Para se representar uma visão de desenvolvimento se utiliza um diagrama como um gráfico de nós ligados por associações de comunicação. Os nós contem componentes que por sua vez, podem conter objetos aos quais indicam que objetos fazem parte dele. Um nó é um objeto físico em tempo de execução que representa os recursos de processamento, geralmente tendo o mínimo de memória e frequentemente a capacidade de processamento. Eles podem ser conectados por associação a outros nós, a qual indica o caminho de comunicação entre eles. A associação pode ter um estereótipo para indicar a natureza do caminho da comunicação (por exemplo, o tipo de canal ou rede). Na visão demonstrada na Figura 10, existem quatro camadas físicas representadas por camada cliente, web, componentes de negócio e informações corporativas. A camada cliente é responsável pela apresentação tanto no browser (comunicando-se com a camada web) como em aplicações desktop ou aplicações clientes Java (que se comunicam diretamente com a camada de negócio). A camada web que roda como um servidor web lidando com requisições e respostas dos clientes invocando JEE

servlets ou Java *Server Pages* (JSP). Camada de Componentes de Negócio que compreendem a lógica de negócio para a aplicação e a camada de sistemas de informações corporativas, que consiste de uma ou mais base de dados e aplicação de retaguarda como mainframes e sistemas legados (Bass, 2006).

Figura 10 – Visão de Desenvolvimento da arquitetura JEE em múltiplas camadas



Fonte: Bass, 2006

2.3.3.2 Estrutura de Implantação

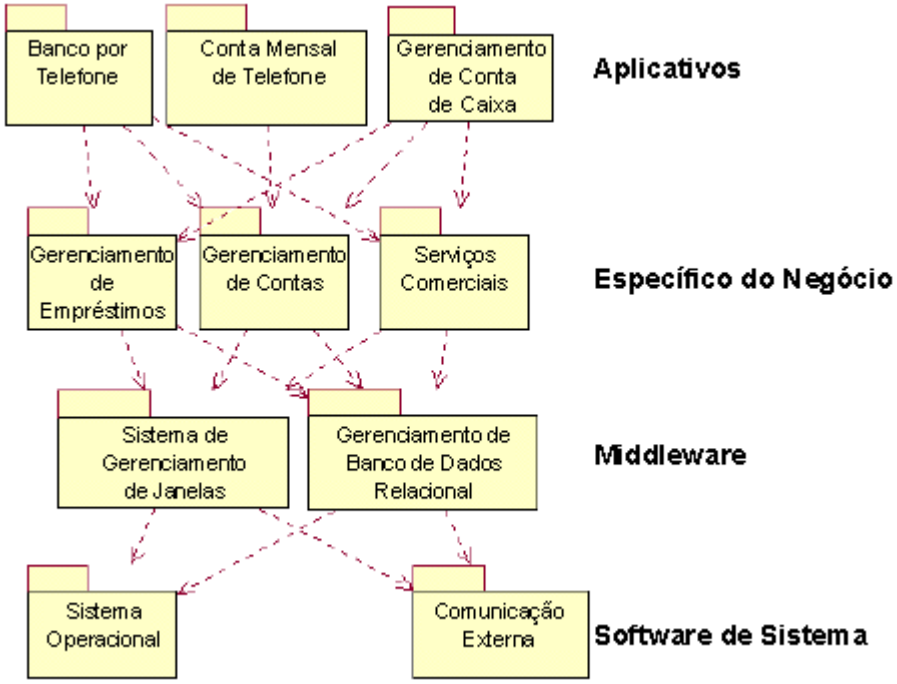
A estrutura de implantação se preocupa em como os elementos de *software* serão mapeados na estrutura de arquivos no desenvolvimento do sistema. Isso é crítico para as atividades de desenvolvimento e construção de processos (Bass, 2006).

Kruchten (2000) descreve a visão de implantação como uma organização estática dos módulos do *software* como: código fonte, arquivos de dados, componentes, executáveis, entre outros. Referindo-se a um ambiente de desenvolvimento seriam pacotes e camadas já em um ambiente de configuração, como propriedades e estratégias de liberação.

Segundo Bass (2006) a estrutura de implantação exibe, se a arquitetura está em conformidade com as decisões de design que foram descritas por ela. Significa que a visão de implantação deve ser divida nos elementos prescritos, aos quais devem interagir uns com os

outros cumprindo com as suas responsabilidades, conforme foi descrito na arquitetura. Normalmente a visão de implantação é representada em camadas, sendo que estas camadas podem variar de acordo com as exigências da aplicação. Na Figura 11 é demonstrado um sistema de banco com quatro camadas. A camada superior contém serviços específicos dos aplicativos. A camada de negócio contém a lógica de negócio que pode ser utilizada em diversos aplicativos. A camada middleware contém componentes como construtores de interface com o usuário, interfaces com banco de dados, serviços de sistemas operacionais, entre outros. A camada mais inferior diz respeito a componentes como sistemas operacionais e interfaces para hardware específico (RUP).

Figura 11 – Visão de Implantação de um sistema Bancário



Fonte: RUP, 2002

2.3.3.3 Atribuição de Trabalho

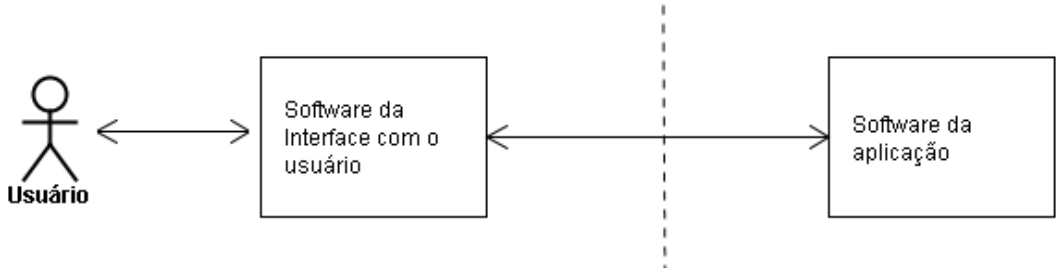
Esta estrutura tem a responsabilidade de atribuir as equipes apropriadas a implementação e integração dos módulos. Também em grandes projetos de desenvolvimento distribuído pode-se utilizar a estrutura de atribuição de trabalho para designar o desenvolvimento de certos tipos de funcionalidades especificas a apenas uma única equipe, ao

invés de tê-las implementadas por todas que precisam dela. Esses tipos de funcionalidades compreendem programas e dados que são privados e certamente subdivido em equipes de trabalho. Nesses casos o arquiteto deverá possuir conhecimento suficiente a respeito das competências de cada equipe, a fim de atribuir corretamente o trabalho a ser executado (Bass, 2006).

2.4 ARQUITETURA DE *SOFTWARE* PARA SISTEMAS INTERATIVOS

Sistemas Interativos requerem uma equipe de projeto atuando em uma diversidade de tarefas estruturadas em um processo. Um sistema interativo pode ser decomposto em duas grandes partes de *software*: *software* de aplicação e *software* de interface com o usuário. O *software* de aplicação diz respeito a toda funcionalidade do sistema enquanto que a interface com o usuário tem a responsabilidade de mediar à comunicação entre o usuário e a aplicação. A decomposição de um sistema interativo é elucidada na Figura 12 (Mendes, 2002).

Figura 12 – Decomposição de um sistema interativo



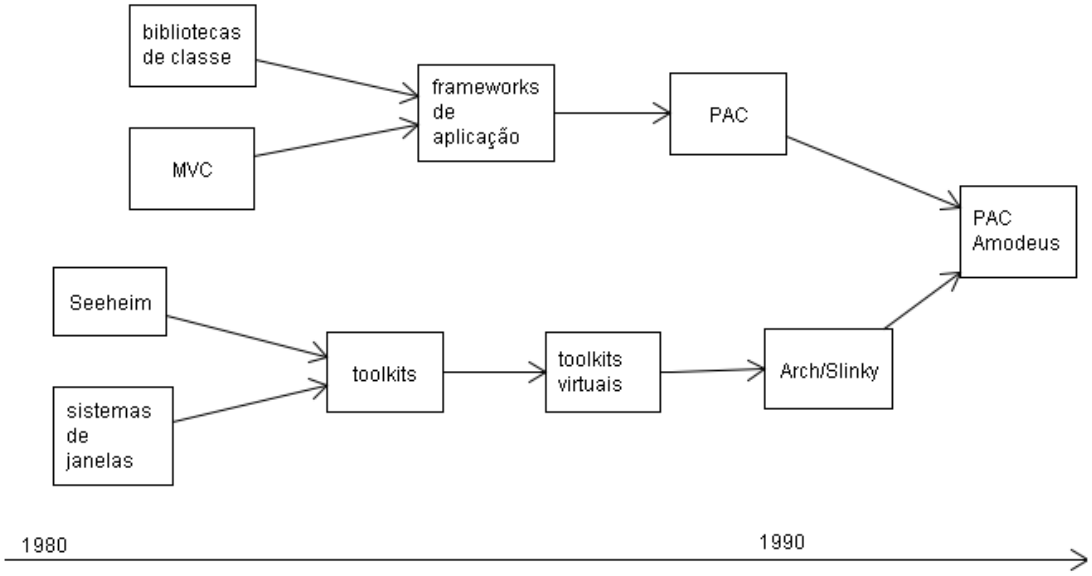
Fonte: Mendes, 2002

Para Pressman (2006), muitos autores sugerem um projeto de arquitetura que desacople a interface do usuário do comportamento da aplicação com o argumento que manter a interface, aplicação e navegação separadas simplifica a implementação e aumenta o reuso. Para ele a arquitetura Modelo-Visão-Controlador (MVC) é uma das arquiteturas que possuem esse princípio.

Mendes (2002) cita que o desenvolvimento dessas arquiteturas coincidiram com o desenvolvimento de diversos tipos de ferramentas (bibliotecas de classe do MVC) e de sistemas de janelas (Arquitetura Seeheim). Os frameworks de aplicação derivaram de um extensão lógica de bibliotecas de classes, além de toolkits e toolkits virtuais. Esses duas

variantes de desenvolvimento evoluíram até chegar numa junção chamada de PAC-Amodeus. O processo evolutivo das arquiteturas de *softwares* interativos é ilustrado na Figura 13.

Figura 13 – Evolução das arquiteturas e *software* de interfaces com usuário



Fonte: Mendes, 2002

A seguir serão apresentadas as principais arquiteturas de sistemas interativos segundo Mendes (2002), além de outras arquiteturas com menor importância que também serão discutidas.

2.4.1 MVC (Modelo Visão Controlador)

Segundo Fowler (2006) o MVC é um dos padrões mais citados. Surgiu como um framework desenvolvido por Trygve Reenskaug para a plataforma Smaltalk no final dos anos 70. A partir de então tem tido um papel influente na maioria dos frameworks com interface para usuário.

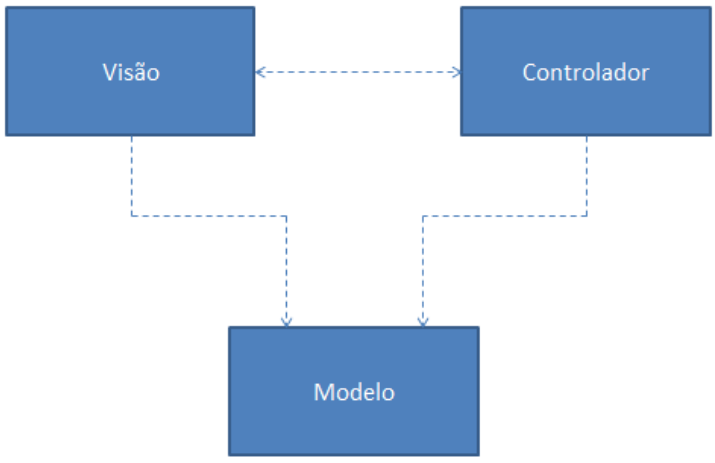
Mendes (2002) menciona que MVC é um modelo baseado em agentes. Os agentes tem um estado, possuem conhecimento bem como, são capazes de inicializar ou reagir a eventos.

Na arquitetura MVC um agente é visto sobre três perspectivas funcionais, Figura 14:

- Modelo – Este modelo representa o conhecimento do domínio da aplicação e contém os componentes do sistema que fazem o trabalho real. Por exemplo, o comportamento do aplicativo principal de um sistema de finanças pessoais com as contas, clientes, transações.

- Visão – Dependendo do usuário essa camada pode prover diferentes perspectivas. Portanto devem oferecer distintas visões do modelo. Tal visão atua como um filtro destacando alguns atributos para um usuário e suprimindo para outro. Por exemplo, um usuário iniciante não possui as mesmas informações que um usuário administrador. No exemplo do aplicativo de finanças, enquanto uma visão mostra uma listagem de uma determinada conta na outra exibe apenas um gráfico pizza. Em ambas as visões usam os mesmos dados, entretanto exibindo de formas diferentes.
- Controlador – O controlador atua como uma interface entre o modelo associado com a sua visão e os dispositivos de interface do usuário. Na atual interpretação é considerado mediador entre o modelo e a visão. Fazendo com que a visão seja responsável pelos dispositivos de entrada e saída e o controlador por mudanças de estado exibição e as ações na interface do usuário e vice-versa. Por exemplo, no aplicativo de finanças pessoais, executando uma rotina na interface do usuário vai refletir nos dados desse aplicativo. Da mesma forma uma alteração da base de dados por aplicativos externos também irá alterar a interface do usuário (Larman, 2007).

Figura 14 – Modelo Visão Controlador



Fonte: Fowler, 2006

Fowler (2006) considera duas separações principais no padrão MVC: separar a apresentação do modelo e separar o controle da vista.

a) **Separar a apresentação do modelo** – é umas das mais importantes e fundamental para um bom projeto de *software* por algumas razões, principalmente por se referirem a preocupações diferentes. Quando se está desenvolvendo uma visão, a preocupação é construir uma boa interface com o usuário, já quando se está trabalhando no modelo o pensamento é com as políticas de negócio, inclusive com interações com banco de

dados. Além disso a separação da apresentação do modelo permite desenvolver interfaces completamente diferentes, o mais notável disso é que o mesmo modelo poderia ser fornecido para clientes ricos, navegador web, desktop e mesmo dentro de uma única interface web poderiam ter diferentes páginas de clientes em diferentes pontos de uma aplicação.

b) **Separar a vista do Controlador –** Fowler (2006) considera uma divisão menos importante. Isto seria útil quando o sistema possuísse uma visão com dois controladores. Porém na prática a maioria dos sistemas tem apenas uma visão por controlador, dessa forma esta separação não era implementada. Ela se torna útil para sistemas com interface web.

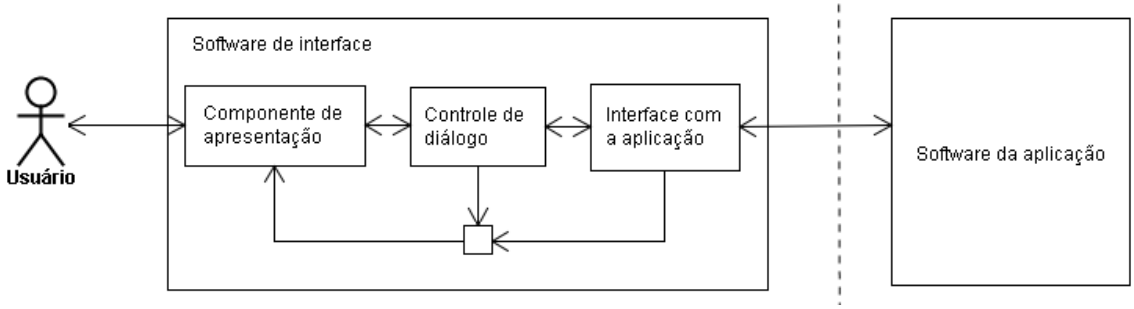
O componente controlador no MVC utiliza um mecanismo de notificação para assegurar que as mudanças no modelo sejam refletidas na visão e vice versa. Como os valores da interface com o usuário mudam estes poderiam mudar os dados no modelo. Além disso, ações de usuário na interface certamente mudarão o comportamento do modelo, e por sua vez também implicaria em mudança nos dados do modelo. Quando uma mudança é percebida na interface esta deveria ser percebida em todas as visões de modelo e não somente na visão que inicializou a mudança. O modelo assim notifica todas suas visões, que por sua vez podem consultar o modelo por informações e atualiza-las se necessário (Fowler, 2006).

2.4.2 Arquitetura Seeheim

Segundo Mendes (2002), vários autores citam a arquitetura Seeheim como uma das mais conhecidas arquiteturas de interface. A arquitetura Seeheim é composta de três componentes: apresentação, diálogo e interface de aplicação. O componente de controle de diálogo tem a responsabilidade do processamento e sequenciamento do diálogo, em sua essência é um organizador do tráfego entre os componentes de apresentação e os componentes específicos da aplicação. O componente de interface com a aplicação possui uma visão da interface sob a perspectiva da aplicação e uma visão da aplicação sob a perspectiva da interface. Isso quer dizer que é encarregada de tratar a comunicação existente entre a interface e o restante do sistema interativo. O componente de apresentação lida com a representação da interface com os usuários, isto é, dispositivos de entrada e saída, layout de tela, entre outros. Pode se notar Figura 15 que demonstra a arquitetura Seeheim que existe uma pequena caixa

que possibilita um desvio da informação desviando o fluxo para outro caminho. Isso pode ocorrer para desviar o fluxo do componente de controle a fim de melhorar o desempenho. (Mendes, 2002).

Figura 15 – Arquitetura Seeheim



Fonte: Mendes, 2002

A arquitetura Seeheim tem a vantagem de abordar explicitamente o problema de independência de diálogo, contudo tem uma desvantagem que é a complexidade dos três componentes em sistemas de grande porte (Mendes, 2002).

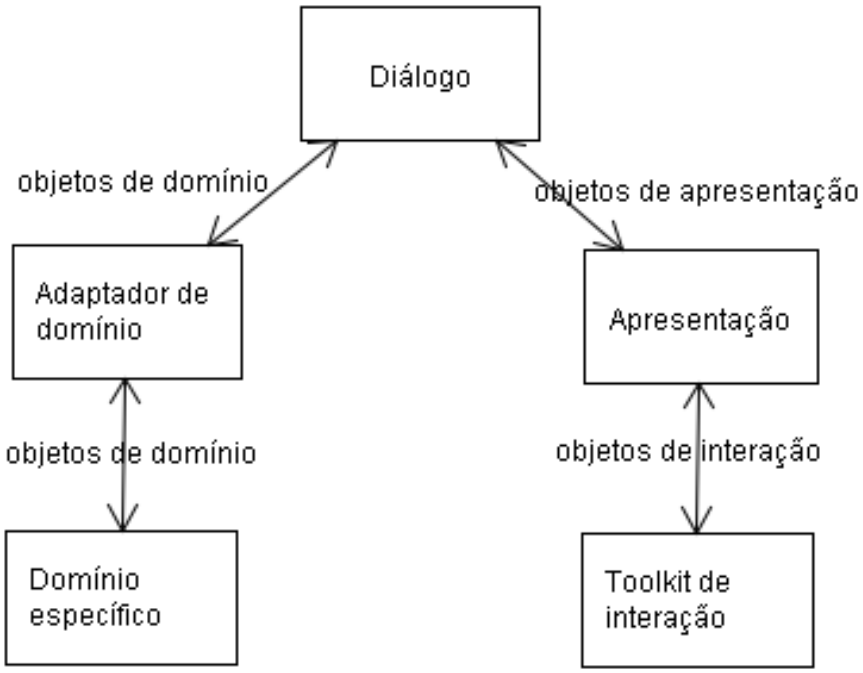
2.4.3 Arquitetura Arch/Slinky

Para Mendes (2002) esta arquitetura pode ser vista como uma extensão da arquitetura Seeheim. Ela é composta por cinco elementos que serão vistos a seguir e é ilustrada na Figura 16.

a) **Componente específico de domínio –** É responsável pela manipulação dos dados e outras funções orientadas ao domínio. As funções são expostas ao usuário através da interface (Mendes, 2002).
b) **Componente adaptador de domínio –** Agrega dados específicos de domínio em estruturas de níveis mais elevados, faz a checagem semântica dos dados e aciona tarefas de diálogo do domínio e relata erros semânticos. É similar à interface na arquitetura Seeheim (Mendes, 2002).
c) **Componente de diálogo –** É responsável por mediar componentes específicos do domínio com componentes específicos da apresentação. Além disso, assegura a consistência entre múltiplas e visões e sequenciamento de tarefas (Mendes, 2002).

d) **Componente de apresentação –** Fornece um conjunto de objetos para o componente de diálogo e pode ser visto como um componente lógico de interação (Mendes, 2002).
e) **Componente do toolkit de interação –** Responsável por implementar interação física entre o usuário e o computador, lida com entradas e saídas e pode ser chamado de componente físico de interação (Mendes, 2002).

Figura 16 – Arquitetura Arch/Slinky



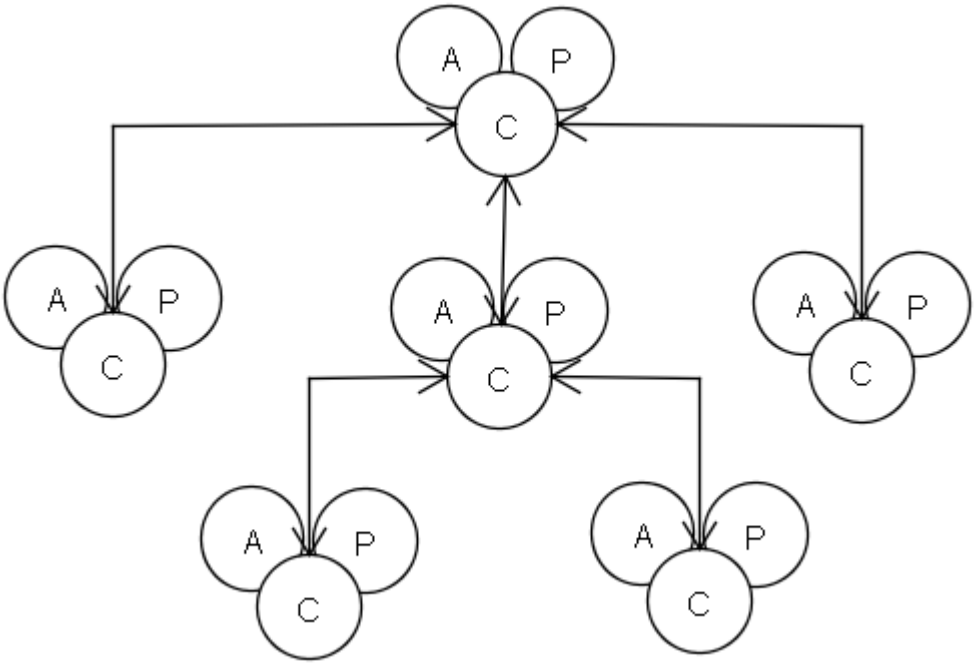
Fonte: Mendes, 2002

A arquitetura Arch/Slinky tem a meta de estabelecer um equilíbrio entre o desenvolvimento e as funcionalidades exigidas por um sistema. Essa arquitetura foi projetada objetivando minimizar efeitos de futuras modificações no toolkit de interação, diálogo de interface com o usuário e domínio de aplicação, além da atribuição de funções heterogêneas a componentes distintos para que qualquer modificação cause o mínimo impacto possível a outros componentes do sistema. Com tudo isso essa arquitetura não atende a alguns critérios como: desempenho do sistema, simplicidade conceitual, reuso de código, atendimento de requisitos funcionais, qualidade da interface com o usuário, entre outros (Mendes, 2002).

2.4.4 Arquitetura PAC

O conceito de arquitetura PAC (*Presentation-Abstraction-Control*) pode ser entendido como um conjunto de agentes PAC que são relacionados uns aos outros através de uma estrutura hierárquica. Cada tríplice constitui em um agenda fortemente acoplado como pode ser visto na Figura 17. Cada agente possui uma apresentação relacionada ao comportamento perceptível de entrada e saída, uma abstração, isto é, seu núcleo funcional e um controle expressando múltiplas formas de dependência (Mendes, 2002).

Figura 17 – Arquitetura PAC



Fonte: Mendes, 2002

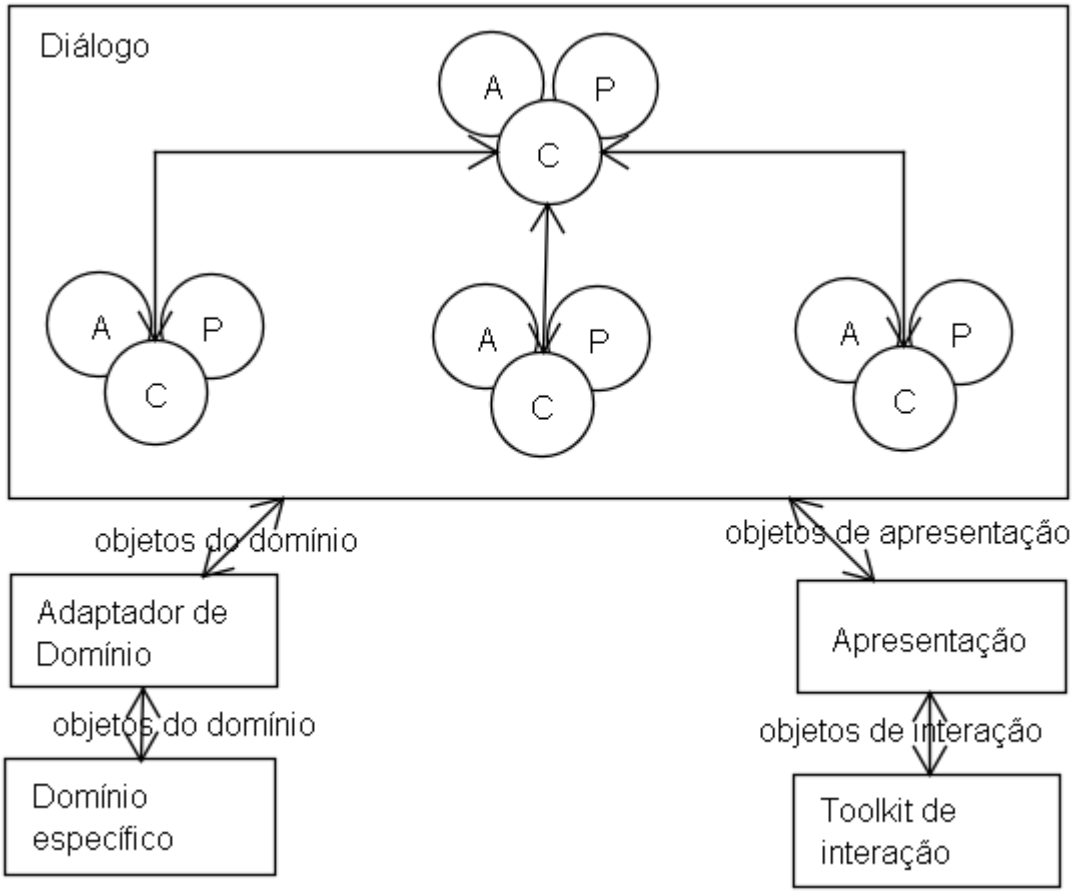
Segundo Mendes (2002), enquanto que a arquitetura MVC desacopla as técnicas de entradas e saídas, a arquitetura PAC as concentra no aspecto de apresentação.

2.4.5 Arquitetura PAC-Amodeus

Para Mendes a arquitetura PAC-Amodeus é considerada híbrida, pelo fato da arquitetura PAC estar embutida na arquitetura Arch/Slinky. Utilizando como base para

decomposição funcional a arquitetura Arch/Slinky conforme mostra a Figura 18. Nesse caso o componente de diálogo da arquitetura Arch/Slinky é povoado de agentes PAC (Mendes, 2002).

Figura 18 – Arquitetura PAC - Amodeus



Fonte: Mendes, 2002

A arquitetura PAC-Amodeus procura unir as melhores características das arquiteturas Arch/Linky e PAC resultando na arquitetura mostrada na Figura 18. Da mesma forma que na arquitetura Arch/Linky ela fornece um fluxo bidirecional de informações entre os componentes primários. No componente de diálogo existem dois fluxos de informações. O primeiro é uma travessia hierárquica dos agentes PAC e o segundo a comunicação direta com o componente adaptador de domínio e o componente lógico de interação com o componente de apresentação (Mendes, 2002).

2.5 PADRÕES DE PROJETO

Projetar um software orientado a objetos não é uma tarefa simples, projetar um software reutilizável orientado a objetos é ainda mais complexo. Um projeto deve ser específico para o problema a ser resolvido, mas também genérico o suficiente para atender problema e requisitos futuros. (Gamma, 2000). Para Gamma (2000) projetistas experientes sabem que não devem resolver cada problema partindo do zero, eles encontram uma boa solução e a utilizam repetidamente. Em consequência disto se encontrará padrões, de classes e de comunicação entre objetos, que reaparecem frequentemente em projetos orientados a objetos (Gamma, 2000). Esses padrões visam resolver problemas específicos dos projetos orientados a objetos deixando-os mais flexíveis e reutilizáveis. Conforme cita Gamma, "Padrões de Projeto tornam mais fácil reutilizar projetos e arquiteturas bem-sucedidas" (Gamma, 2000 p. 18).

Padrões de Projeto variam na granularidade e no nível de abstração. Existem muitos padrões o que torna necessária sua organização. A Tabela 1 mostra como os padrões de projeto estão organizados. São dois critérios utilizados para a classificação de um padrão de projeto. O primeiro critério é a finalidade, ou seja, o que o padrão faz, os padrões podem ter a finalidade de criação, estrutural ou comportamental. Padrões de criação se preocupam com o processo de criação dos objetos. Padrões estruturais lidam com a composição de classes e ou objetos. Padrões comportamentais caracterizam as maneiras como as classes e objetos interagem e distribuem responsabilidade (Gamma, 2000).

Tabela 1 – Padrões de Projeto

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

Fonte: Gamma, 2000.

O segundo propósito é chamado escopo, especifica se o padrão se aplica primariamente a classes ou objetos. Os padrões para classes lidam com o relacionamento entre as classes e subclasses. Os padrões para objetos lidam com relacionamentos entre objetos que podem ser mudados em tempo de execução e são mais dinâmicos (Gamma, 2000).

Com mais de vinte padrões de projeto no catalogo, pode ser difícil escolher um padrão que trata um problema para um projeto específico. Algumas abordagens são utilizadas para encontrar um padrão de projeto correto, Primeiramente é importante conhecer quais são os padrões e a que fim eles se destinam. Também é importante saber como esses padrões se inter-relacionam, e em posse dessas informações é possível direcionar o desenvolvimento para um padrão ou um grupo de padrões adequados (Gamma, 2000).

Uma vez escolhido o padrão de projeto, chega o momento de aplica-lo. Um ponto importante quanto a aplicação de padrões de projeto é ler o padrão por inteiro. Dando ênfase para a aplicabilidade e consequências, assegurando-se de que o padrão é correto para o seu problema. Outro fator que deve ser levado em conta é o estudo do código, é importante estudar um exemplo concreto do padrão codificado. Outra questão é incorporar o nome do padrão ao nome da classe, isto tornará o padrão mais explicito na implementação. Por exemplo, ao utilizar o padrão Strategy em um algoritmo poderá ter a classe como NomeClasseStrategy. Isto são apenas diretrizes que podem ser seguidas na implementação de padrões de projeto (Gamma, 2000).

Para Gamma (2000) não devem ser utilizados indiscriminadamente, frequentemente eles obtêm flexibilidade e variabilidade, porém isto pode custar algo em termos de desempenho. Para ele um padrão deve ser aplicado se a flexibilidade que ele oferece realmente for necessária (Gamma, 2000).

2.6 FRAMEWORKS

Um *framework* é um conjunto de classes que cooperam entre si construindo um projeto reutilizável para uma determinada categoria de software. O *framework* dita a arquitetura da aplicação, definindo a estrutura geral, sua divisão em classes, classes de objetos, como estas colaboram, e o fluxo de controle. O *framework* predefine os parâmetros do projeto de forma que o projetista ou o desenvolvedor possa se concentrar aos aspectos

específicos da sua aplicação. Em resultado disto é possível construir aplicações mais rapidamente, como também de forma similar. Por outro lado o projetista perde a liberdade já que muitas decisões de projeto já foram tomadas (Gamma, 2000).

Segundo Barreto (2006) um dos principais objetivos da Engenharia de Software é o reuso, através disso, se obtém um aumento da qualidade e redução de esforço no desenvolvimento de software. Frameworks possibilitam o desenvolvimento de aplicações a partir de estruturas mais genéricas.

A utilização de frameworks traz diversos benefícios a um projeto, por serem modulares, extensíveis, reusáveis e também por assumirem o controle da execução do sistema a chamada inversão de controle (Barreto, 2006).

O desenvolvimento através de um framework traz além de facilidades alguns desafios, como a curva de aprendizagem que leva tempo e resulta em custo. Ainda com o uso de frameworks pode se ter uma aplicação com desempenho abaixo do desejado. O que é plausível já que aplicações genéricas tendem a ter um desempenho inferior quando comparado a aplicações específicas. Frameworks precisam ser mantidos de diversas formas, como correção de erros de programação, acréscimo ou remoção de funcionalidades e acréscimos ou remoção de pontos de extensão. Com isso, cabe ao desenvolvedor da aplicação considerar o esforço para se obter os benefícios de uma nova versão do framework (Barreto, 2006).

Conforme Barreto (2006) existem inúmeros frameworks para desenvolvimento java, frameworks de aplicação, frameworks de persistência e frameworks web. Entre os mais conhecidos estão o Struts, o Spring Framework e o JSF (Java Server Faces).

O Struts foi um dos primeiros frameworks web desenvolvido e muito popular. Hoje é mantido pela Apache no Projeto Jakarta. Contudo segundo Barreto (2006) o surgimento de tantos outros foi um indicativo de que ele não foi capaz de satisfazer as necessidades das aplicações. O Struts é um framework maduro e possui um módulo de validação capaz de validar tanto no lado cliente quanto no lado servidor. Porém força um grau elevado de acoplamento entre a camada de apresentação e o framework web. O framework Struts pode ser integrado com o Spring Framework (Barreto, 2006).

O JSF pertence a especificação do Java, é um framework baseado em componentes que são usados na camada de apresentação (Barreto, 2006). Este framework possui diversos componentes padrões para a construção de interface com o usuário, mas também permite a utilização de componentes de terceiros. Tem um custo de adoção alto já que os componentes

precisam ser desenvolvidos. Além disso, possui um nível médio de acoplamento. O JSF também pode ser integrado com Spring Framework (Mann, 2005).

O Spring Framework fornece como diz sua própria referência (springframework.org) uma solução leve para construir aplicativos empresariais, possibilidade do uso de gerenciamento de transações, recursos de envio de e-mails, variadas opções para persistência dos dados, permite desenvolver aplicativos utilizando POJO (*Plain Old Java Objects*), integração com AOP (*Aspect Oriented Programmation*), além de um ambiente Web MVC bem projetado fornecendo uma excelente alternativa a outros frameworks web (Hemrajani, 2007). É um framework modular, isto é, permite que sejam utilizadas somente partes dele, sem adicionar as partes restantes (Hemrajani, 2007). A Tabela 2 traz um comparativo entre os frameworks citados.

Tabela 2 – Vantagens e Desvantagens entre os Frameworks citados

Frameworks	Vantagens	Desvantagens
Struts	• Validação tanto em cliente quanto em servidor. • Muito popular por ter sido um dos primeiros. • Disponibiliza conjunto de bibliotecas tags para facilitar o desenvolvimento de aplicações baseadas em HTML.	• Força um elevado grau de acoplamento entre a camada de apresentação e o framework web. • Exige criar classes específicas para representar dados inseridos em formulários, mesmo quando um modelo é mais adequado.
JSF	• Permite que componentes de interface sejam criados e reaproveitados. • Existe no mercado uma série de componentes pronto e gratuitos. • Desenvolvimento de páginas com RAD (*Rapid Application Development*)	• Custo de adoção alto, pois exige que os componentes sejam criados. • Não possui validação do lado cliente. • Grau de acoplamento superior ao Spring framework.
Spring Framework	• Permite trabalhar com POJOS o que o torna leve. • Singletons, elimina a necessidade do desenvolverdor manter classes Singleton próprias. • Framework Web. • Gerenciamento de Transações. • *Container* leve. • Fraco grau de acoplamento. • Diversos componentes.	• Curva de aprendizado alta. • Configurações em XML se tornam difíceis. • Não fornece validação do lado cliente.

Fonte: adaptado pelo autor

A seguir o Spring Framework será apresentado em detalhes.

2.6.1 Spring Framework

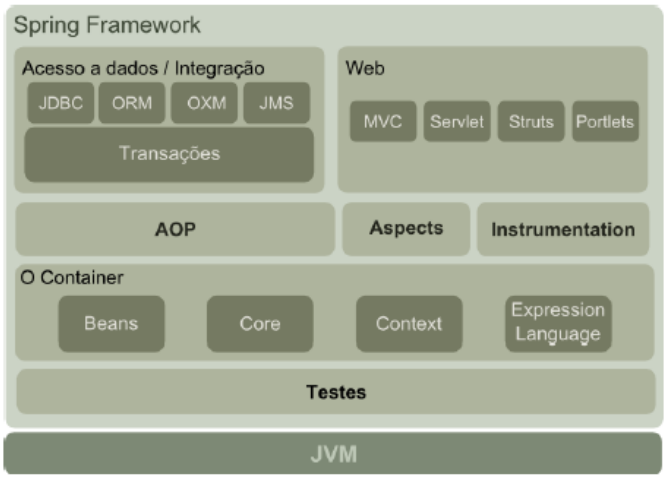
O Spring é um framework de código aberto, que foi criado originalmente por Rod Johnson para lidar com a complexidade do desenvolvimento de aplicações corporativas, tornando possível a utilização de JavaBean o que antes era possível somente com EJB´s (Enterprise Java Bean). Ele é baseado no conceito de inversão de controle e injeção de dependência, onde o controle da aplicação é invertido e o *framework* controla o fluxo de execução das tarefas. Hoje o Spring é mantido por uma divisão da VMware (Weissmann, 2012).

O Spring não se limita ao desenvolvimento do lado do servidor, qualquer aplicação Java pode se beneficiar do Spring em termos de simplicidade, testabilidade e baixo acoplamento. Ele faz muitas coisas, mas na essência ele simplifica o desenvolvimento Java (Barreto, 2006). Segundo Walls (2011) muitos frameworks pretendiam simplificar uma coisa ou outra, mas o Spring visa simplificar como um todo o desenvolvimento Java. O Spring simplifica o desenvolvimento a partir de quatro estratégias chave: (Walls, 2011)

- Desenvolvimento leve e minimamente invasivo a partir de POJO´s (*Plain old Java objects*).
- Diminuindo o acoplamento através de injeção de dependência e inversão de controle.
- Programação declarativa através de anotações e convenções comuns.
- Reduz a repetição de código através de aspectos e *templates.*(Walls, 2011)

Em sua versão básica, o Spring é formado por seis componentes, o *container* de inversão de controle, suporte a AOP (*Aspect Oriented Programmation*), instrumentação, acesso a dados/integração, suíte de testes e web, como mostra a Figura 19.

Figura 19 – Componentes básicos do Spring



Fonte: Walls 2011.

Um dos principais diferenciais do Spring no seu lançamento foi trazer recursos de computação corporativa onde antes eram oferecidos somente por servidores de aplicação pesados. Além disso, ao contrário de um container EJB (*Enterprise Java Bean*), não é uma solução tudo ou nada, podem ser utilizados somente os componentes que interessam ao seu projeto, ignorando o que não é necessário e como resultado disso, tem-se uma arquitetura mais leve e fácil de trabalhar (Weissmann, 2012).

2.6.1.1 Container

O *Container* é o único módulo obrigatório para se trabalhar com Spring. Fazem parte do núcleo do *framework* os módulos *core* e *beans*, neles são implementados o suporte à inversão de controle e injeção de dependências. No Spring existem dois tipos de *containers* o BeanFactory que existe desde a primeira versão do *framework* que é basicamente um versão sofisticada do padrão *Factory*. O segundo recurso é o *AplicationContext* baseado no BeanFactory que além da IOC (Inversion of Control) e DI (Dependency Injection), oferece recursos mais avançados como gerenciamento de recursos e internacionalização (Weissmann, 2012).

Por fim o módulo SpEL (*Spring Expression Language*), que é a forma mais popular de configuração do Spring através de arquivos XML. A SpEL é uma linguagem muito parecida a utilizada com JSPs (Java Server Pages), porém voltada para configuração do *container* (Weissmann, 2012).

2.6.1.2 AOP (Aspect-Oriented Programming)

O Spring fornece um suporte muito abrangente para Programação orientada a Aspectos em um módulo de AOP. Este módulo serve como base para o desenvolvimento de seus próprios aspectos. Assim como o DI, a AOP se concentra no desacoplamento dos objetos da aplicação (Walls, 2011).

2.6.1.3 Instrumentação

A equipe de Spring também se preocupou com o que vem a acontecer após o *deploy*. O módulo de Instrumentação oferece facilidades que auxiliam a equipe de manutenção permitindo o acompanhamento em tempo de execução tudo o que acontece com o sistema, gerando diversas estatísticas através da implementação de JMX (Java Management Extensions) (Weissmann, 2012).

2.6.1.4 Acesso a dados e Integração

O Spring fornece um módulo de acesso a dados com suporte as principais tecnologias de persistência adotadas por desenvolvedores Java, como JDBC, ORMs como Hibernate, JPA e OXM. O suporte a estas tecnologias se da através de *templates* que reduzem significativamente a quantidade de código, nesse caso é necessário apenas abrir e fechar conexões, iniciar e finalizar transações.

Um dos pontos importantes deste módulo é o controle transacional, que vai além do que é oferecido por servidores de aplicação.

Além do suporte a frameworks de persistência que foram informados, e o controle transacional, este módulo possui um suporte nativo a JMS (*Java Message Service*) que permite a comunicação entre dois sistemas através de mensagens. (Weissmann, 2012).

2.6.1.5 Web

O Modelo-Visão-Controle (MVC) é o paradigma comumente utilizado na construção de uma aplicação *web* para separar a interface de usuário da lógica da aplicação (Walls, 2011).

Segundo Walls (2011) a linguagem Java possui inúmeros frameworks MVC, entre os mais populares estão o Apache Struts e JSF. Apesar do Spring integrar com esses frameworks, ele possui em sua gama de módulos, um módulo MVC capaz de promover técnicas de baixo acoplamento na camada web de uma aplicação (Walls, 2011).

Segundo Weinssmann (2012) o projeto Spring MVC não foi planejado pela equipe de desenvolvimento do *framework*, ele surgiu conforme eles foram aceitando o fato de que o Struts (*Framework* que dominava o mercado) apresentava uma série de limitações que dificultavam a separação de interesses entre as camadas de controle, negócio e visualização (Weinssmann, 2012).

2.6.1.6 Testes

Reconhecendo a importância dos testes no desenvolvimento de uma aplicação, o Spring prove um módulo dedicado a testar aplicações. Com este módulo o desenvolvedor tem a sua disposição uma coleção de implementações de objetos simulados para escrever unidades de testes que funcionam com JNDI (*Java Naming and Directory Interface*), s*ervlets* e *portlets* (Weissmann, 2012).

2.6.1.7 Portfólio do Spring

Para Weinssman (2012) o Spring *Framework* é tão versátil que em pouco tempo após o seu lançamento, a entidade responsável pelo Spring desenvolveu para ele uma série de componentes opcionais. Dentre os mais famosos está o Spring Security, que é responsável pela segurança da aplicação de uma maneira extremamente robusta e podendo ser aplicado em qualquer projeto Spring.

Outro componente é o Spring *Data* que é utilizado para lidar com bases de dados relacionais ou não e abstrai do desenvolvedor detalhes que se tornam repetitivos de implementação, através de *templates* que facilitam a integração com dados.

Ainda existem outros componentes como o Spring *Social* para integração com as redes sociais mais consagradas e também para quem trabalha com processamento em lote existe o Spring *Batch*, entre outros componentes (Wiensmann, 2012).

2.7 MÉTODOS DE ANÁLISE ARQUITETURAL

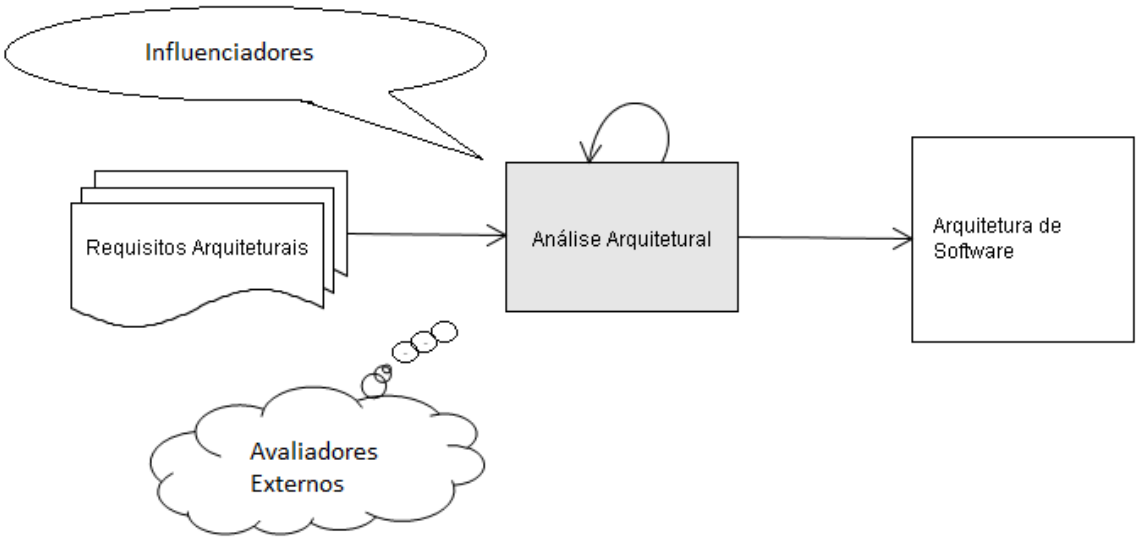
No desenvolvimento de uma arquitetura se chega num estágio em que é preciso se discutir como avaliar ou analisar a arquitetura, para se certificar que o *model*o produzido fará o trabalho desejado. Essa análise pode responder questão sobre avaliação arquitetural como: Por que, quando, custo, benefícios, técnicas, planejada ou não, pré-condições e resultados (Bass, 2006).

Existem diversas abordagens para análise de sistemas de *software* para Mendes (2002). Segundo ele algumas análises geralmente englobam as seguintes atividades:

- **Desenvolvimento de modelos arquiteturais** – a coleta dos dados é realizada durante a elicitação dos requisitos sendo analisados e incorporados ao *model*o arquitetural.
- **Melhoria e síntese de uma solução –** Incrementa as informações descritas no *model*o arquitetural.
- **Análise da solução –** Análise da solução realizada, podendo identificar a necessidade de refinar o modelo.

Uma questão importante segundo Mendes (2002) é que o processo de análise de uma arquitetura de *software* é um processo iterativo, ao qual são feito refinamentos de um modelo de arquitetura inicial derivado de um conjunto de requisitos arquiteturais, ou seja, propriedades desejadas ou apresentadas por uma arquitetura de *software*. O processo de análise arquitetural pode ser visto na Figura 20.

Figura 20 – Análise Arquitetural



Fonte: Mendes, 2002

A avaliação produzirá um relatório ao qual todos os domínios de interesse juntamente com os dados de suporte devem ser descritos. O relatório deve circular na fase de projeto para todos os avaliadores com o intuito de capturar e corrigir qualquer equívoco antes de ser finalizado (Bass, 2006).

Existem alguns métodos que são utilizados para a avaliação de arquiteturas de *software* já na fase de iniciação, serão citados três métodos: SAAM (*Software* Architecture Analysis Method), ATAM (Architecture Trafeoff Analysis Method) e CBAM (Cost Benefit Analysis Method).

2.7.1 SAAM (*Software* Architecture Analysis Method)

Este método foi inicialmente concebido para o auxílio a arquitetos de *software* com o objetivo de realizar comparações arquiteturais. A proposta inicial do SAMM consistia em:

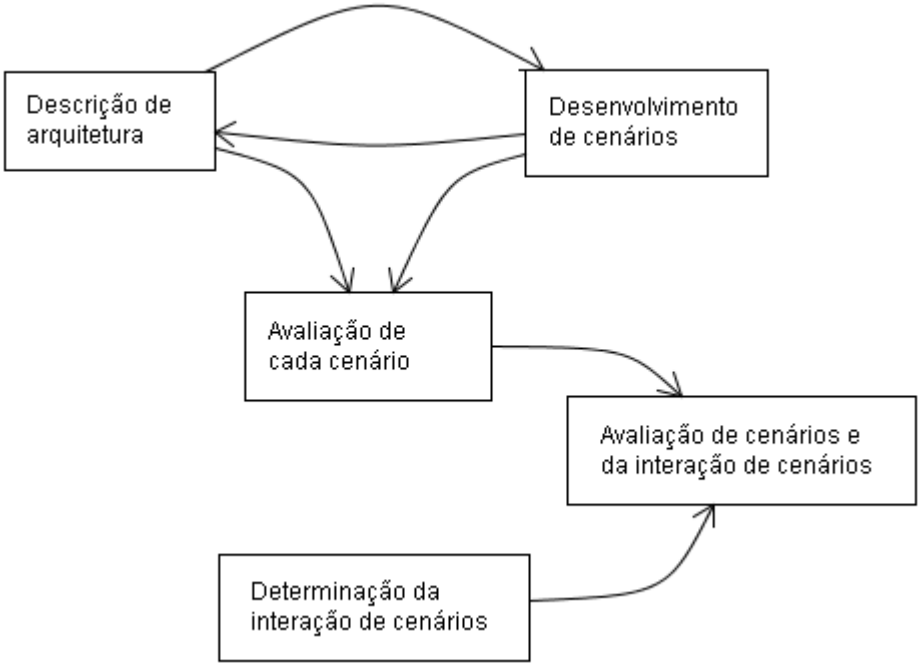
1. Definição de cenários que representem um aspecto importante do sistema no domínio, incluindo pontos de vista das pessoas que possuíam papeis de influenciadores no sistema.
2. Utilização dos cenários para gerar uma partição funcional do domínio, uma ordenação parcial das funções e um acoplamento dos cenários às várias funções existentes na partição.

3. Usar a partição funcional e ordenação parcial juntamente com os cenários de uso para realizar a análise das arquiteturas propostas.

Cada cenário definido obtém uma pontuação das arquiteturas que foram avaliadas conhecidas como arquiteturas candidatas. O avaliador tem a responsabilidade de determinar pesos para os cenários, assim como as pontuações globais das arquiteturas candidatas (Mendes, 2002).

Esse método tem como base o uso de cenários, dessa forma o analista pode avaliar o potencial do sistema de *software* e prover suporte a atributos de qualidade. O método SAAM compreende um conjunto de cinco passos que são interdependentes entre si, conforme ilustra a Figura 21 (Mendes, 2002).

Figura 21 – Etapas de SAAM



Fonte: Mendes, 2002

1. **Desenvolvimento de cenários –** os cenários de tarefas devem ser desenvolvidos de uma forma que ilustrem os tipos de atividades que o sistema deve prover suporte.
2. **Descrição da arquitetura –** Cada análise deve possuir uma representação arquitetural do sistema, fazendo uso de uma notação arquitetural definida, descrevendo componentes e conectores utilizados na especificação da arquitetura.
3. **Avaliação de cada cenário –** A partir de cada cenário deverá ser determinada se a arquitetura candidata suporta ou não as tarefas contidas nos cenários.

4. **Determinação da interação de cenários –** A interação entre os cenários ocorre quando dois ou mais cenários indiretos exigem modificações em algum componente ou conexão, isto é mostra tarefas em que os componentes estão associados.
5. **Avaliação de cenários e da interação de cenários –** É uma etapa subjetiva que envolve os influenciadores do sistema. É realizada uma avaliação global atribuindo escores a cada cenário (Mendes, 2002).

2.7.2 ATAM (Architecture Tradeoff Analysis Method)

O método ATAM é assim chamado porque ele revela o quão bem uma arquitetura satisfaz as metas de qualidade determinadas, e fornece uma compreensão sobre como interagir metas de qualidade (Bass, 2006).

Nesse método os avaliadores precisam entender a arquitetura, reconhecer os parâmetros arquiteturais que estão sendo utilizados, conhecer as implicações deste parâmetros em relação aos atributos de qualidade e comparar estas implicações com os requisitos do sistema (Mendes, 2002).

Segundo Bass (2006) o ATAM requer uma participação e cooperação mutua de três grupos:

1. Equipe de Avaliação – este grupo é externo ao projeto cuja arquitetura está sendo avaliada, tornando as decisões sobre a arquitetura imparciais. A cada membro da equipe são atribuídas funções específicas a desempenhar durante a avaliação, conforme mostra a
2. Tabela 3.
3. Tomadores de decisão do Projeto – Essas pessoas tem poder para falar do projeto de desenvolvimento ou tem autoridade para ordenar mudanças.
4. Interessados na arquitetura – Os interessados na arquitetura tem interesse em saber se a mesma está sendo realizada conforme foi anunciada.

Para Bass (2006) o método ATAM deverá produzir as seguintes saídas:

- Apresentação concisa da arquitetura
- Articulação das metas de negócio
- Requisitos de qualidade em termos de um conjunto de cenários
- Mapeamento das decisões da arquitetura aos requisitos de qualidade

- Um conjunto de pontos e de sensibilidade identificado nos pontos de escolha.
- Um conjunto de riscos e não riscos
- Um conjunto de temas de riscos

Estas saídas são utilizadas para produzir um relatório final que recapitula o método, sumariza o processo, captura os cenários e suas análises e catálogos dos resultados (Bass, 2006).

Tabela 3 – Papeis da equipe de avaliação

Papel	Responsabilidade	Características Desejáveis
Líder da Equipe	Define-se a avaliação, coordenadas com o cliente, fazendo com que suas necessidades sejam satisfeitas, estabelece contrato de avaliação, forma a equipe de avaliação e analisa o relatório final e entrega.	Bem organizada, com capacidade de gestão, boa interação com o cliente, capaz de cumprir os prazos.
Líder de Avaliação	Executa avaliação, facilita a elicitação de cenários, administra processo de cenário seleção/priorização, facilita a avaliação de cenários contra arquitetura.	Confortável em frente ao público, excelentes habilidades de facilitação, boa compreensão das questões arquitetônicas, pratico nas avaliações da arquitetura, capaz de dizer quando uma discussão está levando a uma descoberta ou quando é inútil e deve ser redirecionada.
Descritor de Cenário	Escreve cenários sobre flipchart ou quadro branco durante elicitação de cenário, captura as concordâncias sobre redação de cada cenários, parando a discussão até que a formulação exata seja capturada.	Boa caligrafia, defensor da ideia de não ir em frente antes que um cenário seja capturado, pode absorver e destilar a essência de discussões técnicas.
Descritor de Processo	Captura processos em formato eletrônico, investiga os cenários, gera uma lista de cenários adotados para todos os participantes.	Bom e rápido datilógrafo, bem organizado para a rápida recuperação de informações, boa compreensão das questões arquitetônicas, capaz de assimilar questões técnicas.
Cronometrista	Ajuda o líder de avaliação quanto à programação e os tempos aplicados em cada cenário.	Disposto a interromper uma discussão a fim de não extrapolar o tempo estimado.
Observador do Processo	Mantém notas sobre o processo de avaliação para ser melhorado ou desviado, pode fazer discretas observações baseadas em sugestões de processos para o líder de avaliação.	Atento observador, conhecedor do processo de avaliação, deve ter experiência no método de avaliação de arquitetura.
Executor do Processo	Ajuda o líder a lembrar de executar os passos do método de avaliação.	Fluente nas etapas do método e estar disposto a fornecer orientações ao líder de avaliação
Interrogador	Levantar questões de interesse arquitetural que os interessados não pensaram.	Bons conhecimentos de arquitetura, boas ideias sobre as necessidades dos interessados, experiência com sistemas de domínios similares.

Fonte: Bass, 2006

Alguns benefícios que são listados pela SEI (Kazman, 98) a partir da utilização do método ATAM podem ser vistas a seguir:

- Promove a coleta de requisitos específicos de qualidade.
- Cria um início precoce na documentação de arquitetura.
- Cria uma base documentada para decisões de arquitetura.
- Promove a identificação de riscos no início do ciclo de vida.
- Incentiva o aumento da comunicação entre as partes interessadas.

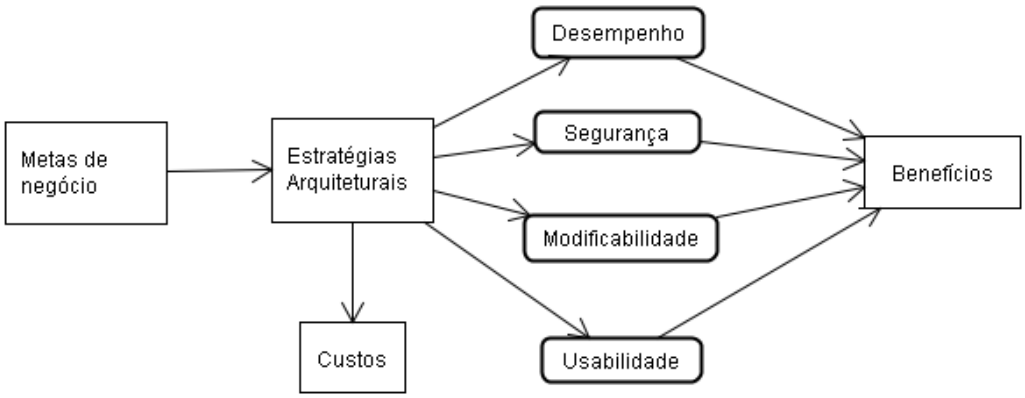
2.7.3 CBAM (Cost Benefit Analysis Method)

O método ATAM fornece um meio de avaliar os pontos de escolha técnicos enfrentados durante a concepção ou a manutenção de um sistema de *software*. No ATAM prima por investigar o quão bem a arquitetura real ou proposta vou bem projetada respeitando os atributos de qualidade que os interessados consideraram importante. São também analisados os pontos de escolha (ocorre quando se abre mão de algo para se obter outro) da arquitetura, onde uma decisão pode ter consequências para uma série de atributos de qualidade simultaneamente (Bass, 2006).

Contudo no ATAM faltam importantes considerações. O maior ponto de escolha em grandes e complexos sistemas geralmente tem a ver com fatores econômicos. Como deveriam ser organizados os recursos investidos de um modo que maximize o ganho e minimize os riscos? No passado a questão mais importante segundo Bass (2006) eram os custos, hoje tão importante ou talvez mais, são os benefícios que uma decisão arquitetural pode trazer para uma organização. O método CBAM constrói sobre o ATAM, para *model*ar os custos e os benefícios de decisões de projeto arquitetural como um meio para aperfeiçoar as decisões. O CBAM fornece uma avaliação das questões técnicas e econômicas e decisões arquitetônicas (Bass, 2006).

Para Mendes (2002), as implicações técnicas compreendem as características do sistema e os atributos de qualidade desejados enquanto que as implicações econômicas referem-se ao custo de implementar o sistema. A Figura 22 demonstra o contexto das decisões arquiteturais e implicações associadas.

Figura 22 – Contexto para o CBAM



Fonte: Bass, 2006

2.8 DOCUMENTANDO ARQUITETURAS DE *SOFTWARE*

A documentação da arquitetura é o passo para se chegar a tudo isso. Para Bass (2006) até mesmo a mais perfeita arquitetura se torna inútil se os interessados não a entenderem. A arquitetura deve ser descrita com detalhes suficientes, sem ambiguidade e organizada de tal forma que seja possível encontrar rapidamente o que for necessário. Além disso, cada interessado no sistema tem diferentes necessidades, a documentação deve satisfazer cada um deles com diferentes tipos, níveis e tratamentos da informação (Bass, 2006).

Bass (2006) cita um exemplo (Tabela 4) dos interessados e as informações que eles esperam encontrar na documentação.

Segundo Bass (2006), um dos conceitos mais importantes associado à documentação de uma arquitetura de *software* são as Visões Arquiteturais. Elas são uma representação coerente das estruturas arquiteturais, escritas e lidas pelos interessados no sistema. Para documentar uma arquitetura deve-se escolher as visões mais relevantes. Nesse momento o arquiteto deve conhecer os interessados e o uso que eles planejam fazer da documentação, sendo que cada um deles vai utiliza-la para um propósito.

Tabela 4 – Interessados e as necessidades de comunicação servidos pela arquitetura

Interessado	Uso
Arquiteto e Engenheiros de Requisitos que representam o cliente	Negociar e fazer compensações entre os requisitos concorrentes
Arquiteto e projetistas de partes constituintes	Resolver a contenção de recursos e estabelecer desempenho e outros tipos de orçamento de recursos de consumo em tempo de execução
Implementadores	Fornecer restrições invioláveis nas atividades de desenvolvimento
Testadores e Integradores	Especificar um correto comportamento caixa preta das partes que devem trabalhar juntas
Administradores	Revelar as áreas que as mudanças poderão afetar
Projetistas de outros sistemas que este deverá operar	Definir um conjunto de operações providas e requeridas e os protocolos das operações
Especialistas em Atributos de Qualidade	Prover um *model*o que deverá conter a informação necessária para a avaliação de um variedade de atributos de qualidade
Gerentes	Para criar equipes de desenvolvimento com as aptidões necessárias, para planejar e alocar recursos ao projetos, bem como acompanhar o progresso das diversas equipes
Gerentes de Linha de Produto	Determinar se um membro em potencial ou uma nova família de produtos está dentro ou fora do escopo.
Equipe de Qualidade	Prover uma base para checar a conformidade, garantindo que o que está descrito na arquitetura foi realmente implementado

Fonte: Bass, 2006

A Tabela 5 representa um certo número de interessados e as visões que são úteis a cada um. Nesta tabela estão as três estruturas utilizadas da Figura 3, Visões modulares, Visões de Componentes e Conectores (C&C) e Visões de Alocação.

Tabela 5 – Interessados e as visões arquiteturais mais úteis

	Visões Modulares				Visões C&C	Visões de Alocação	
Interessados	**Decomposição**	**Uso**	**Classe**	**Camadas**	**Várias**	**Desenvolvimento**	**Implementação**
Gerente de Projeto	pi	pi		pi		d	
Desenvolvedores	d	d	d	d	D	pi	pi
Testadores e Integradores		d	d				
Administradores	d	d	d	d	D	S	pi
Construtor de Produto (Build)		d	pi	O	pi	S	pi
Cliente					pi	m	
Usuário Final					pi	pi	
Analista de Sistemas	d	d	pi	d	pi	d	
Analista de Infraestrutura	pi	pi		pi		pi	d
Arquiteto	d	d	d	d	D	d	pi

d= Informação Detalhada, pi = Pouca informação m = Informação Macro

Fonte: Bass, 2006

2.9 CONSIDERAÇÕES FINAIS

O desenvolvimento de uma arquitetura de software envolve uma série de decisões, que se tomadas mais cedo em um projeto de software, diminuem a complexidade e custo das mudanças. A arquitetura de software é trabalhada em diversas fases do processo unificado. Na fase de iniciação tem como resultado a definição de uma arquitetura candidata Para a definição de uma arquitetura candidata é importante conhecer as estruturas arquiteturais, que juntas compõem o software como um todo, afim de se obter a qualidade desejada através do atributos que são explicados através da norma ISO/IEC 9126.

Tabela 6 – Vantagens e limitações de arquiteturas de sistemas interativos

Arquitetura de *Software*	Vantagens	Limitações
Monolítica	É um *model*o de implementação, poderia ser usado para obter eficiência, mas não é recomendado.	Não há separação entre interface e aplicação e não é recomendada para sistemas interativos.
Cliente-Servidor	Impõe a forma de comunicação entre cliente e servidor.	Não há decisão clara das decisões do projeto entre o cliente e o servidor.
Seeheim	Suporta a modificação e suporte à portabilidade.	Requer modificações no componente de diálogo quando o toolkit é substituído.
MVC	Além dos atributos do Seeheim, distribui o estado da interação entre agentes e suporta a modularidade. É extremamente popular no desenvolvimento para WEB e utilizada por diversos frameworks de desenvolvimento.	Dificuldade em projetar uma arquitetura com base em agente devido à ausência de um conjunto explícito de níveis de abstração como encontrado no Arch/Slinky. Homogeneidade já que os aspectos são representados num único estilo. Resolvido em PAC-Amodeus.
Arch/Slinky	Suporte à portabilidade e facilidade de modificação.	Camadas adicionais afetam o desempenho.
PAC	Vide MVC. Suporte às modificações no diálogo. A hierarquia PAC é composta por um grande número de componentes de diálogo.	Vide MVC. Não é fácil efetuar modificações na apresentação, pois este componente é distribuído entre os agentes "folha" na hierarquia PAC.

Fonte: Mendes, 2002

Com o advento da interatividade do usuário com o *software* e a exigência cada vez maior de interfaces cada vez mais amigáveis, ganhou importância as interfaces que separam a lógica da aplicação com a própria interface com o usuário. São as chamadas arquiteturas de interface. As arquiteturas de interface trazem características diferentes tornando-as apropriadas para determinadas aplicações. Cada uma delas apresenta vantagens e desvantagens conforme mostrou a Tabela 6 (Mendes, 2002).

Projetar um software com alto grau de complexidade e com todas as variações existentes não é tarefa fácil. Nem os projetistas mais experientes tenta um resolver um problema partindo do zero. Eles encontram uma solução e a utilizam repetidamente, surgindo

os chamado Padrões de Projeto. Existem inúmeros padrões de projeto e para fins variados. Uma maneira de aplicar esses padrões é com a adoção de um *framework*.

Frameworks já possuem várias decisões de projeto implementadas e com elas alguns padrões de projeto que já estão implantados. Eles se baseiam no conceito de Inversão de Controle, ou seja, tomam para si o controle de alguns pontos da aplicação, deixando o desenvolvedor preocupado apenas com questões inerentes a lógica da sua aplicação. Apesar de terem uma curva de aprendizado alta, compensam pela padronização que exigem do código e também pelo aumento da produtividade já que possuem diversos *templates* que facilitam o desenvolvimento.

Após todas as escolhas exigidas para se chegar a um produto de qualidade e atendendo aos atributos de qualidade determinados, é preciso avaliar ou analisar a arquitetura candidata se certificando que o modelo produzido realmente fará o trabalho desejado. Como forma de verificar se a arquitetura candidata está de acordo com o que foi proposto podem ser aplicados métodos de análise arquitetural, as quais darão subsídios para os projetistas ou arquitetos para uma avaliação mais consciente das decisões que serão tomadas, avaliando riscos, novos cenários e atributos de qualidade ainda não atendidos.

Por fim, mesmo uma excelente arquitetura se não for bem documentada não terá utilidade. Como existe um número grande de interessados no sistema e esses com objetivos diferentes. Dessa forma se faz necessária uma documentação coerente e que traga a cada um deles informações com objetivo de elucidar o funcionamento do *software* e como os componentes devem ser implementados no hardware disponível (Bass, 2006).

3. DEFINIÇÃO DA ARQUITETURA CANDIDATA PARA O NUSIS

Entre umas das tarefas da fase de iniciação do software está a definição de uma arquitetura candidata. A arquitetura candidata servirá como base para o desenvolvimento do software, deve ser um processo iterativo e evoluir ao longo do projeto.

O projeto do NUSIS é caracterizado por uma equipe inexperiente com uma alta rotatividade por ser formado maioritariamente por alunos. Os *softwares* a serem desenvolvidos são considerados complexos e devem ser integrados entre si, e com outros softwares.

Para apoiar a escolha da arquitetura candidata e podendo ser aplicado durante todo o projeto do NUSIS propõe-se o método de análise arquitetural ATAM. Este método é mais completo e uma evolução de seu precursor, o método SAAM ambos criados pela SEI. Inclui pontos de escolha (*tradeoffs*) como sua principal característica considerando aspectos técnicos e financeiros de um projeto. Seu objetivo é ajudar a escolher uma arquitetura adequada para um sistema de *software*, descobrindo pontos de escolha e pontos de sensibilidade e minimizar o risco destas escolhas (Bass, 2006).

O ATAM foi projetado visando participação das partes interessadas para focar a atenção dos avaliadores sobre a parte da arquitetura que é fundamental para a realização dos objetivos (Bass, 2006). O ATAM compreende quatro fases macro, explicadas na Tabela 7.

Tabela 7 – Fases do ATAM

FASE	ATIVIDADE	DESCRIÇÃO DA FASE
Fase 0	Parceria e Preparação	Os líderes da equipe de avaliação e os principais tomadores de decisão se reúnem para discutir o trabalho. Os representantes do negócio resumem o trabalho a fim de selecionar as pessoas que possuam conhecimentos adequados. Por fim o líder da equipe explica quais as informações o gerente e o arquiteto deverão apresentar na fase 1 e os ajuda caso necessário.
Fase 1	Avaliação	A equipe de avaliação estudará a documentação da arquitetura e terá ideias sobre o sistema e abordagens gerais arquitetônicas e os atributos de qualidade que são importantes.
Fase 2	Continuação da Avaliação	Os interessados participam do processo e da análise continua.
Fase 3	Acompanhamento	A equipe de avaliação produz e fornece um relatório final.

Fonte: SEI

Compondo as quatro fases informadas estão os nove passos para a realização do ATAM, explicados na Tabela 8.

Tabela 8 – Nove passos do ATAM

	Passos do ATAM	Tarefa	Objetivos
Fase 1	Passo 1	Apresentar o ATAM	Apresentação do ATAM aos representantes do projeto.
	Passo 2	Apresentar fatores do negócio	Apresentar uma visão geral das perspectivas do negócio.
	Passo 3	Apresentar a arquitetura	O arquiteto faz a apresentação da arquitetura em um apropriado nível de detalhes.
Fase 2	Passo 4	Identificar abordagens arquiteturais	Catalogar os padrões e abordagens que são evidentes.
	Passo 5	Gerar os atributos de qualidade na Árvore de Utilidade	Os atributos de qualidade são articulados em detalhes por um mecanismo chamado Árvore de Utilidade, tornando os requisitos concretos a partir dos cenários. Existe a priorização de cenários.
	Passo 6	Analisar as abordagens arquiteturais	Os cenários mais ranqueados são avaliados um a um.
Fase 3	Passo 7	Brainstorm e priorização de cenários	A equipe de avaliação debate os cenários, verificando se as ideias dos interessados batem com as ideias do arquiteto.
	Passo 8	Analisar as abordagens arquiteturais	A equipe repete a avaliação realizada no passo 6, mapeando os cenários recém gerados.
Fase 4	Passo 9	Apresentar resultados	Apresentar os resultados resumindo as informações coletadas mais uma vez para as partes interessadas.

Fonte: SEI

Os tópicos a seguir descrevem o processo da escolha da arquitetura candidata.

3.1 **PRINCIPAIS FATORES DE NEGÓCIO**

Todos os envolvidos na avaliação necessitam saber, incluindo os membros da equipe de avaliação o contexto do sistema e os principais fatores do negócio que motivaram o desenvolvimento (Bass, 2006).

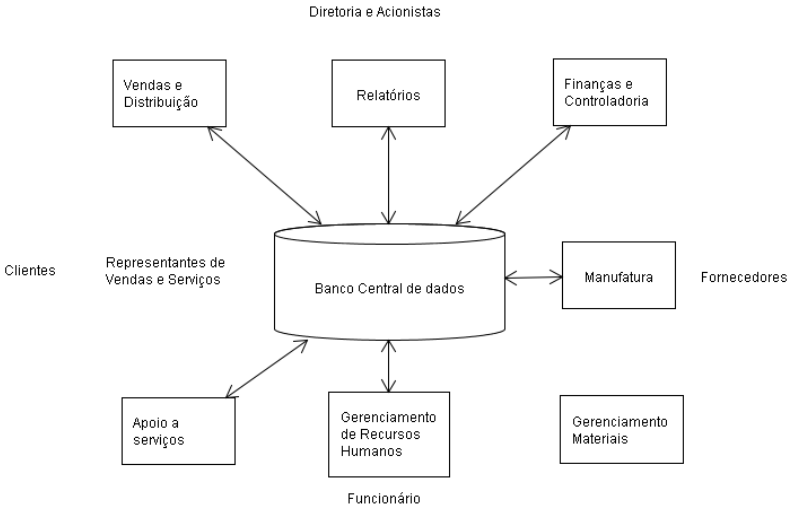
O NUSIS possui um Projeto chamado “Sistemas nas Empresas”, que visa oferecer sistemas a empresas de pequeno porte. Existem diversos sistemas que serão desenvolvidos por alunos juntamente com os professores do núcleo do CCTI, também poderão ser envolvidos alunos e professores de outras áreas de conhecimento. Entre os diversos sistemas a serem desenvolvidos, está um ERP baseado na Web em linguagem Java. Existe uma premissa para o projeto em que todos os componentes utilizados devem ser *Softwares* Livres, resultando na diminuição dos custos para o NUSIS e para as empresas.

A partir destas informações foram levantadas características importantes em relação à arquitetura que sistemas ERP e WebApps possuem, para se chegar nos atributos de qualidade desejados em um projeto com estas caraterísticas.

3.1.1 ERP do NUSIS

O ERP é um *software* que permite a integração entre os dados dos sistemas de informação transacionais e dos processos de negócio de uma organização. Ele não deve ser desenvolvido para um cliente especifico e sim, seguir as melhores práticas de mercado. Uma empresa que implanta um ERP deve-se adaptar as suas funcionalidades e adequar seus processos à modelagem imposta pelo sistema (Caiçara, 2008). O ERP deve utilizar um banco de dados único como repositório central de dados, seguindo uma estrutura como mostra a Figura 23 (Caiçara, 2008)

Figura 23 – Estrutura típica de um ERP



Fonte: Caiçara, 2008

O ERP do NUSIS será um software baseado na WEB, ou seja, possui muitas características distintas de um software para *desktop*, uma delas é estar disponível na grande rede o que já o torna de certa forma diferente. Os WebApps como são chamados os *softwares* baseados na web, se tornaram partes estratégicas do negócio, tanto para pequenas quanto para grandes organizações. Prova disto é que a maioria das grandes empresas desenvolvedoras de ERP já disponibilizam alguma forma de acesso via web aos seus sistemas.

Para o desenvolvimento do ERP do NUSIS que alia características de um software de gestão com a mobilidade da WEB algumas propriedades são importantes:

a) Banco de dados único – tendo em vista o objetivo principal do ERP que é integrar todas as áreas a arquitetura deverá ser composta de apenas um banco de dados.
b) Estrutura Modular – Deve ser composto por módulos que possam trabalhar independentemente uns dos outros.
c) Melhores Práticas – ao implementar um ERP o fornecedor desse tipo do *software* faz um estudo minucioso a fim de identificar as melhores práticas para cada segmento.
d) Segurança – Já que será baseado na WEB, pode estar disponível para o mundo inteiro através da internet. Fortes medidas de segurança devem ser implementadas para proteger conteúdos reservados e a transmissão de dados.
e) Interoperabilidade – Deve permitir a integração com *softwares* legados ou softwares desenvolvimentos especificamente para o cliente.
f) Desempenho – o usuário desse tipo de *software* não está disposto a esperar por processos lentos do lado do servidor.
g) Disponibilidade – Esse atributo se torna imprescindível na medida em que usuários do mundo inteiro com fusos de horário diferentes podem acessar o sistema.
h) Evolução continuada – os WebApps diferentemente de *softwares* convencionais evoluem continuamente. Dessa forma um atributo como modificabilidade deve ser levado em conta.

Para oferecer ao cliente um *software* de qualidade e que de fato traga vantagens competitivas às organizações que o utilizarem a equipe deve estar capacitada para tal. Por se tratarem de alunos do curso de graduação, estes deverão ser supervisionados pelo corpo docente tanto do CCTI quando dos outros centros envolvidos neste projeto.

3.2 APRESENTAÇÃO DA ARQUITETURA

As características para o ERP do NUSIS foram levantadas e o desenvolvimento de um sistema desse porte deve possuir uma equipe com conhecimento que possa trazer reais vantagens aos clientes que o implantarem. Uma maneira de aumentar a produtividade e ainda

buscar uma padronização de código é com a adoção de um *framework,* já que eles implementam diversos padrões de projeto.

Existe uma série de frameworks no mercado entre os mais utilizados está o Spring Framework. O Spring pode ser utilizado de forma a complementar e não a substituir a arquitetura da especificação Java. O Spring Framework é um framework leve, podendo ser rodado em um servidor de aplicação como o TomCat que não necessita grande capacidade de processamento. Possui um *Container* de inversão de controle e injeção de dependência que visam reduzir o acoplamento entre as classes e definir o ciclo de vida dos objetos da aplicação, além disso, possui grandes facilidades que tornam o desenvolvimento mais rápido e evolui rapidamente, às vezes, mais rápido que a própria especificação do Java. (Weinsmann, 2012).

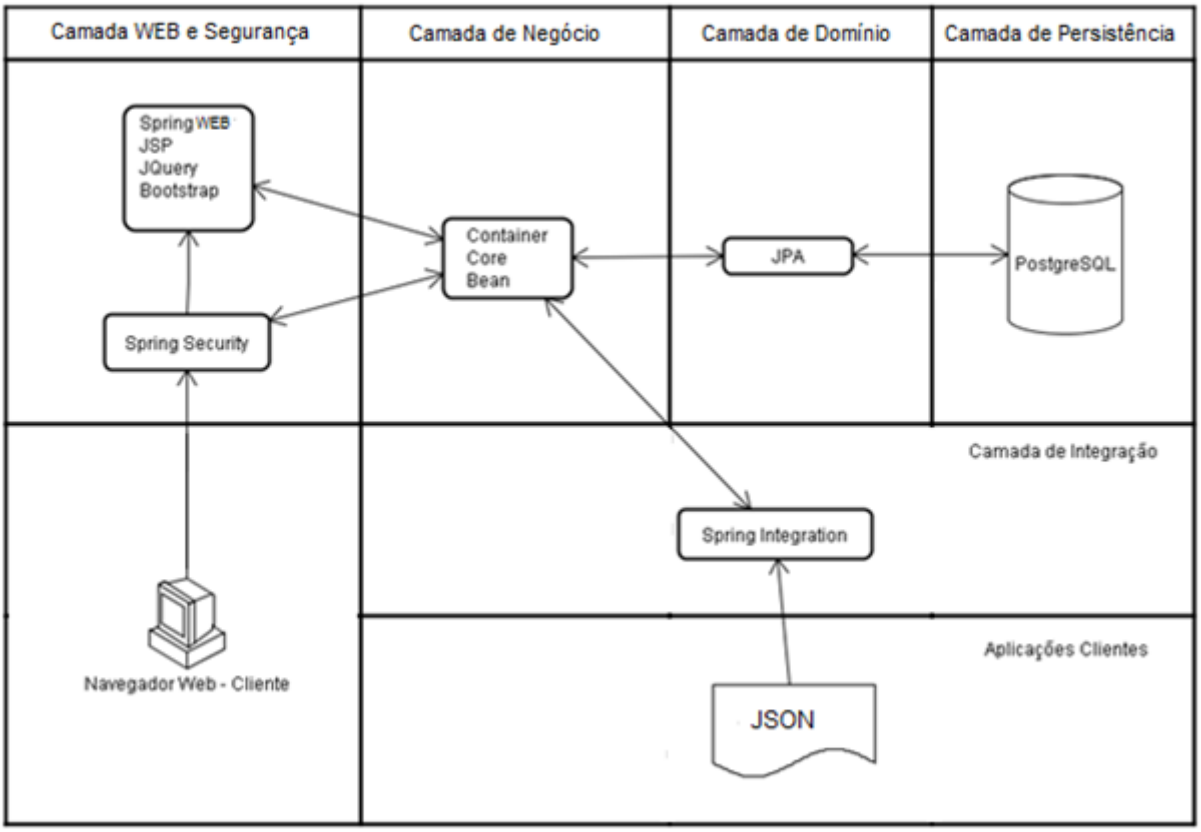
Devido às características do projeto do NUSIS que é atender a empresas de pequeno porte, o Spring *Framework* com sua gama de componentes seria o mais adequado. Por ser uma ferramenta completa tornaria o aprendizado menos complicado, visto que não seria necessária a utilização de tantos outros frameworks. Além disso, por não necessitar de grande capacidade computacional e ser capaz de rodar em máquinas com menos recursos (Hemrajani, 2007). E por fim, mas não menos importante, seria uma forma de padronizar o código.

O capítulo 2 trouxe as estruturas arquiteturais bem como os diagramas para as diversas formas de representa-las. Um software complexo necessita de uma documentação coerente e que possa trazer a equipe clareza nas ideias. Portanto a seguir a arquitetura será demonstrada na forma de diagramas.

3.2.1 Estrutura de Implantação

A estrutura de implantação deve trazer a organização estática dos módulos do software, mostrando como os elementos interagem entre si cumprindo com as suas responsabilidades. A Figura 24 mostra a visão de implantação, a seguir será explicada cada camada bem como os componentes que a compõe.

Figura 24 – Visão de Implantação da arquitetura candidata



Fonte: o próprio autor

3.2.1.1 Camada de Segurança

Na camada de segurança será utilizado o Spring Security é um framework de controle de acesso, o que não pode ser confundido com um framework de segurança. Ele não permite criar uma aplicação invulnerável, a função dele é controlar a autenticação e permissões de acesso. O Spring Security pode ser aplicado em sistemas de duas formas por invocação de métodos dos beans gerenciados pelo contexto do Spring ou pelas requisições que chegam até a aplicação (Walls, 2011).

O Spring Security é composto por diversos módulos praticamente independentes que possibilitam que sejam aplicados no projeto os que forem necessários. Os seguintes módulos poderão ser utilizados no projeto do NUSIS

- **Spring-security-core** – é o único módulo obrigatório, neles estão implementados os mecanismos de autorização e autenticação além das interfaces básicas que serão aproveitadas em todos os módulos desenvolvidos para o framework;

- **Spring-security-config** – se tornou praticamente obrigatório a partir da versão 2.0, por reduzir drasticamente o quantidade de configurador que o desenvolvedor necessita digitar, o que resultará num ganho de produtividade;
- **Spring-security-web** – este é o módulo onde encontram-se toda a infraestrutura necessária. Nesse módulo está presente o código responsável por lidar com a interceptação das requisições, por se tratar de um sistema Web é obrigatório para o projeto do NUSIS.
- **Spring-security-taglibs** – contém uma biblioteca de tags permitindo definir quais áreas das páginas o usuário autenticado no sistema pode acessar com base nas suas permissões.
- **Spring-security-ldap** – suporte a autenticação para servidores LDAP, permitindo integração com o Active Directory, caso o cliente possua esta ferramenta (Weinssmann, 2012).

3.2.1.2 Camada Web

A camada web e segurança do ERP do NUSIS será composta do Spring Web MVC que é o módulo do Spring responsável pela implementação do conceito Modelo-Visão-Controle que foi explicado em detalhes no capítulo 2. E conforme Weissmann (2012) é um padrão muito adotado por frameworks de desenvolvimento Web. O Spring WEB MVC trabalha com um arquivo XML que recebe todas as requisições e tem a função de encontrar as classes controles que por sua vez interagem com o modelo e as visões.

As páginas serão criadas em Java Server Pages (JSP) que segundo a documentação do Java, fornece uma maneira simplificada para o desenvolvimento de aplicações baseadas na web. Possui um conjunto de bibliotecas chamadas JSP Standard Tag Library (JSTL) que implementam funcionalidades de uso em geral, ainda permitem a inserção direta de código Java na própria página.

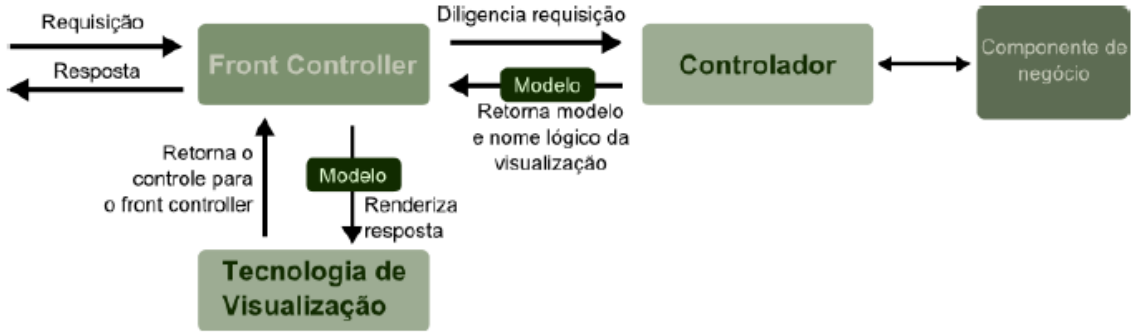
Como frameworks de *front-end* serão utilizados, o Bootstrap para facilitar o desenvolvimento web com uma interface elegante para o usuário, o Modernizr utilizado para detectar recursos de HTML5 e CSS3 nos navegadores dos usuários. Para manipulação dos eventos Ajax será utilizada a API Jquery.

A seguir será explicado o funcionamento do Spring Web MVC em detalhes.

3.2.1.2.1 Spring Web MVC

O Componente responsável pelo funcionamento é o Dispatcher Servlet, que é uma implementação do Padrão Front Controller. Esse padrão tem o objetivo de fornecer um único ponto de entrada central para todas as requisições direcionadas a aplicação. Ele tem a função de decidir qual o componente será o responsável por seu processamento e eventualmente retornar ao usuário. A Figura 25 mostra como este padrão se encaixa no contexto do Spring Web MVC (Spring Source).

Figura 25 – Padrão Front Controller no Contexto do Spring

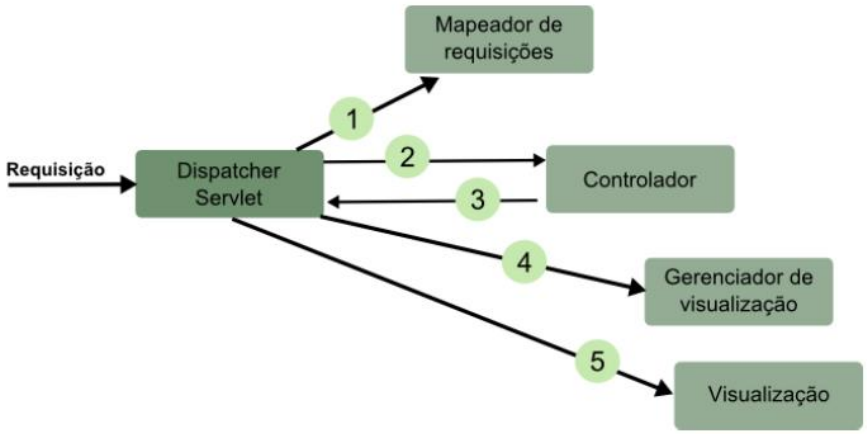


Fonte: springsource.org

Apesar de raramente ser necessário interagir diretamente com este componente, é de fundamental importância conhecer seu funcionamento.

Toda chamada ao servidor inicia sob uma forma de requisição e termina como uma resposta. Entre a requisição e a resposta ocorrem cinco eventos que são elucidados a seguir pela Figura 26.

Figura 26 – Ciclo de Vida de uma requisição



Fonte: springsource.org

Como pode ser visto na Figura 26, tudo começa através de uma requisição que chega no Dispatcher Servlet, que terá sua assinatura analisada e enviada ao Mapeador de Requisições (*Handler Mapping*). Este componente é responsável por descobrir qual controlador deve ser acionado. Depois de obtido o controlador alvo, este é executado e um nome lógico de *template* de visualização, é retornado ao Dispatcher Servlet e o conjunto de variáveis (*Model*) que serão expostas nessa renderização. Com o nome lógico disponível entra em ação o gerenciador de visualização (View Resolver) que retorna ao Dispatcher Servlet uma página JSP que deverá ser renderizada para o usuário que fez a requisição (Walls, 2011).

3.2.1.3 Camada de Negócio

Na camada de negócio está o núcleo do Spring Framework chamado de Container onde estão os dois módulos obrigatórios o *core* e *bean.* Neles são implementados o suporte à inversão de controle e a Injeção de Dependência. O módulo *Bean* utiliza uma implementação sofisticada do padrão de projeto *Factory* o que torna desnecessária a configuração manual de *Singletons* e permite o desacoplamento total do gerenciamento de dependências e ciclo de vida dos *beans* do código fonte através de um mecanismo de configuração (Walls, 2011).

A partir desses padrões implementados pelo Spring Framework, o desenvolvedor pode se preocupar inteiramente com a criação de código específico da aplicação deixando para o Container o controle do ciclo de vida dos objetos.

3.2.1.4 Camada de Persistência

Na camada de persistência será utilizado o Framework da própria especificação Java o Java Persistence API (JPA) que mapeia objetos-relacionais através de POJOS (*Plain Old Java Objects*). Segundo Keith (2009) é viável utilizar JPA na maioria das aplicações corporativas desenvolvidas em Java. O JPA surgiu para facilitar a persistência em linguagens orientadas a objetos com banco de dados relacionais, com ela o desenvolvedor deixa de escrever muitas linhas de código para converter linha e coluna isso se torna transparente a partir desta API (Keith, 2009).

Entre as principais vantagens da utilização do JPA em projeto Java estão:

- Persistência através de POJOS
- Utiliza a linguagem JPQL (Java Persistence Query Language) deixando a aplicação independente de banco de dados.
- Não é intrusiva, ou seja, separa bem os objetos persistentes da camada de persistência.
- Simples configuração, a configuração acontece através de Java *annotations.*
- Fácil de testar é possível escrever o código de persistência em um servidor integrado e ser capaz de reutiliza-lo para testar fora do servidor (Keith, 2009).

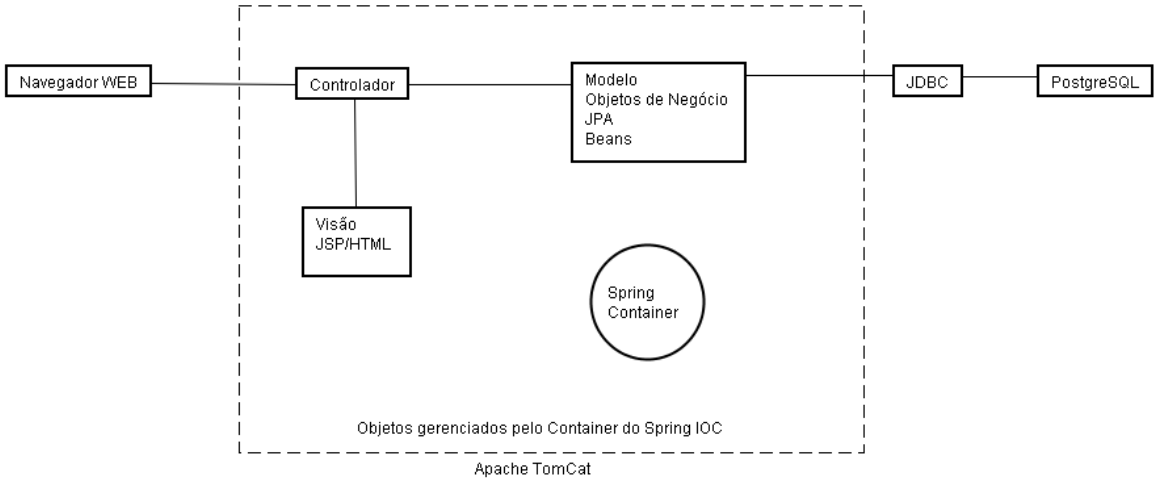
3.2.1.5 Camada de Integração

O Spring Integration é muito utilizado em aplicações corporativas, visto que as mesmas geralmente interagem com outras aplicações corporativas. Esse módulo do Spring oferece implementações de padrões comuns de integração (Wall, 2011). Em outras palavras, as preocupações de mensagens e integração são tratadas pelo framework e os desenvolvedores não necessitam se preocupar com responsabilidades de integração mais complexas (SpringSource). Outro fator importante na escolha do Spring Integration é que a curva de aprendizado desse módulo é baixa já que se baseia nos mesmos conceitos do Spring Core (Wall, 2011). A integração é realizada através de um formato conhecido como JSON (JavaScript Object Notation) que conforme a própria documentação do JSON pode ser utilizada em diversas linguagens como C, C++, Python, entre outras.

3.2.2 Estrutura de Usos

A estrutura de usos demonstra as tecnologias em tempo de execução do *software* que será desenvolvido, visa definir a arquitetura adequada para o Projeto do NUSIS. A Figura 27 traz a estrutura de usos mostrando como essas tecnologias se adaptam para fornecer uma solução em tempo de execução completa.

Figura 27 – Diagrama de Uso da arquitetura candidata



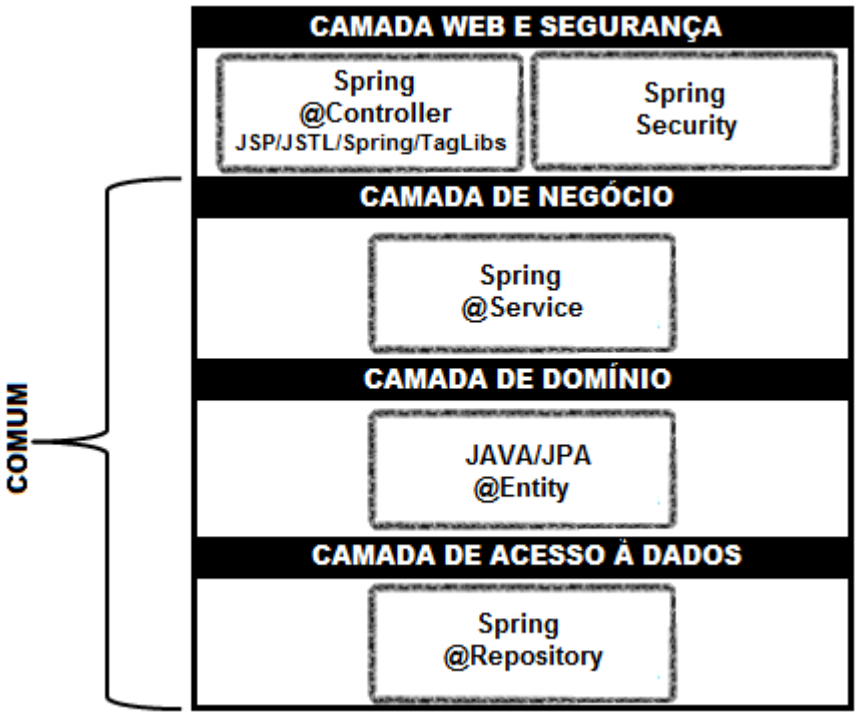
Fonte: adaptado de Hemrajani, 2007

Na Figura 27 é demonstrada como é a organização das classes do Sistema em relação ao Spring. O Controlador recebe as requisições oriundas do navegador Web através de um Servlet chamado de Servlet Dispatcher que trabalha como um Front Controller. Esse controlador requisita uma View e devolve ao usuário conforme sua requisição, após a interação do usuário é acionado o Modelo no qual estão as regras de negócio e persistência de dados. Todo o processo é gerenciado sobre o Container de IOC do Spring rodando em cima do servidor Apache TomCat.

3.2.3 Estrutura em Camadas

Uma camada nada mais é que um conjunto coerente de funcionalidades relacionadas. A Figura 28 mostra como o sistema deverá ser disposto em camadas. Cada camada representa um pacote que irá conter classes com os mesmos interesses. A camada web e segurança estarão presentes os componentes que Controle (@Controller) do Spring bem como as visões e ainda o componente Spring Security que é responsável por capturar qualquer requisição e deve prover a autenticação e autorização de acesso ao sistema. As camadas abaixo da camada WEB, chamadas de Comum, são as classes mais específicas da aplicação e podem ser facilmente reutilizadas em outros projetos Java. Na camada de negócio estão as classes chamadas de *Service* (@Service), responsáveis por executar os procedimentos específicos da aplicação e também possui componentes para integração com outros *softwares*. Na camada de domínio estão as entidades (@Entity), isto é os POJOS que serão persistidos. A camada de acesso a dados onde estarão os repositórios (@Repository) para a manipulação, leitura e escrita dos dados no banco de dados. Todas as classes possuem uma anotação que sempre vem precedida do arroba (@) e são necessárias para a configuração do Spring Framework.

Figura 28 – Visão em camadas da arquitetura candidata

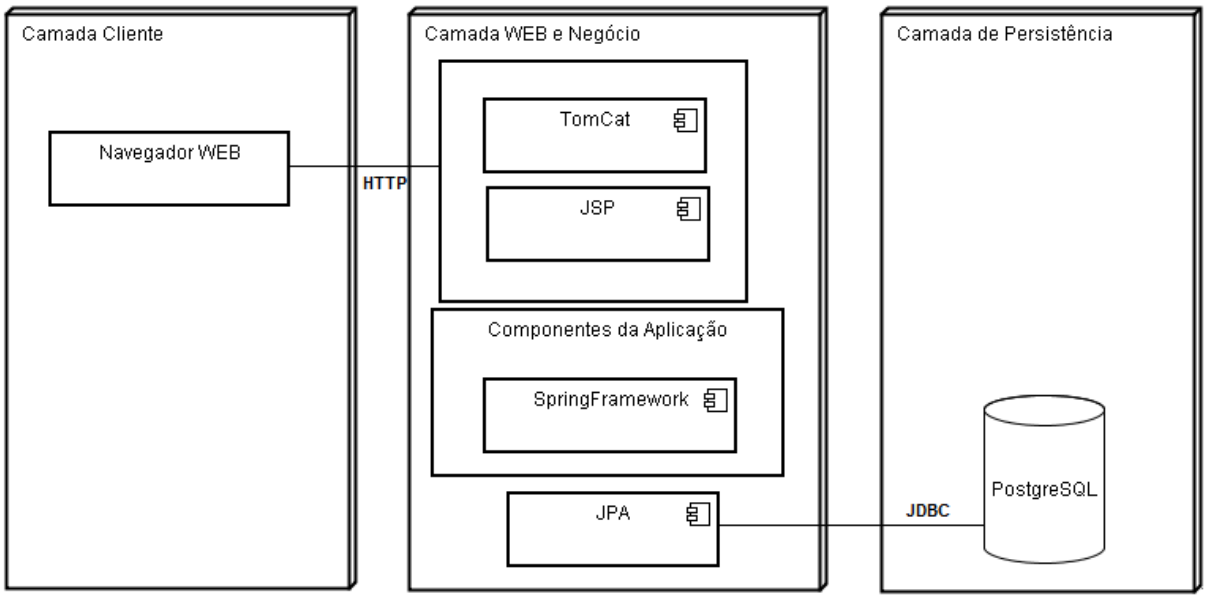


Fonte: o próprio autor

3.2.4 Estrutura de Desenvolvimento

A estrutura de desenvolvimento mostra como os componentes do software devem estar dispostos no hardware e os seus elementos de comunicação. A estrutura de desenvolvimento permite ao engenheiro pensar em como atender aos atributos de qualidade exigidos. A Figura 29 mostra como o software do NUSIS estará disposto no hardware.

Figura 29 – Visão de Desenvolvimento da arquitetura candidata



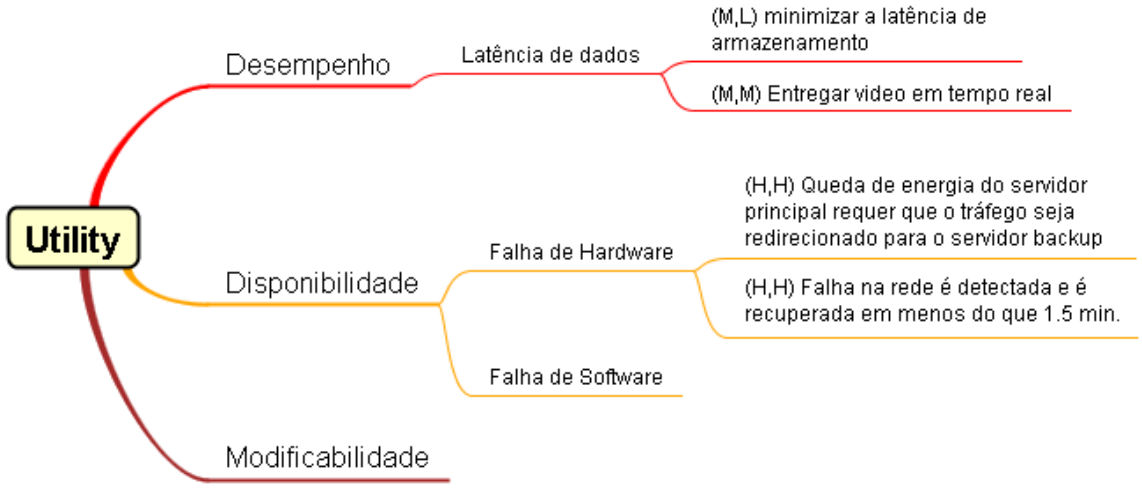
Fonte: o próprio autor

3.3 ATRIBUTOS DE QUALIDADE NA ÁRVORE DE UTILIDADE

As importantes metas dos atributos de qualidade foram nomeados no item 3.2, quando os fatores de negócio foram apresentados. Porém não foram apresentados em detalhes e não permitem análise. Objetivos gerais como alto desempenho e modificabilidade estabelecem um contexto e direção importantes, mas não são suficientemente específicos para nos dizer o bastante sobre a arquitetura. Nesta etapa os atributos de qualidade são articulados em detalhes por um mecanismo conhecido como Árvore de Utilidade. Aqui a equipe de avaliação trabalha junto aos tomadores de decisão para identificar, priorizar e refinar as metas mais importantes dos atributos de qualidade, os quais são expressos como cenários. A Árvore de utilidade serve para tornar os requisitos concretos. A Árvore de Utilidade (Figura 30) inicia com um nodo

principal chamado “Utility”. Os atributos de qualidade (nomeados no item 3.2) formam um segundo nível. Tipicamente desempenho, modificabilidade, disponibilidade são filhos de utility. Para cada um desses atributos de qualidade é realizado um refinamento, isto é, com um maior nível de detalhes a partir dos cenários criados previamente (Bass, 2006).

Figura 30 – Árvore de Utilidade



Fonte: o próprio autor

Os cenários são os mecanismos para os quais os atributos de qualidade desejados são especificados e testados. Eles formam as folhas da árvore de utilidade agrupados pelos atributos de qualidade que expressam. O método ATAM consiste em escolher um cenário por vez e analisar o quão bem a arquitetura responde ou atinge os atributos de qualidade. Alguns cenários podem expressar mais do que um atributo de qualidade e podem aparecer em mais de um lugar na árvore. Isto não é necessariamente um problema, mas o líder de avaliação deve evitar cenários que cubram um território muito grande. Neste caso deve-se tentar dividir o cenário em componentes com menos responsabilidades.

A árvore de utilidade contém uma etapa de priorização. Por consenso os tomadores de decisão atribuem uma prioridade para cada cenário. Pode-se utilizar uma escala de 1 a 10 ou a notação *High*(H), *Medium*(M) e *Low*(L). A segunda opção é a recomendada por Bass (2006), pois é difícil chegar a números precisos. Cada cenário tem um par associado como por exemplo: (H,H), (H,M). O artefato gerado pela árvore de utilidade é uma lista priorizada de cenários que servem como um plano para o restante da avaliação (Bass, 2006).

3.3.1 Cenários de Atributos de Qualidade

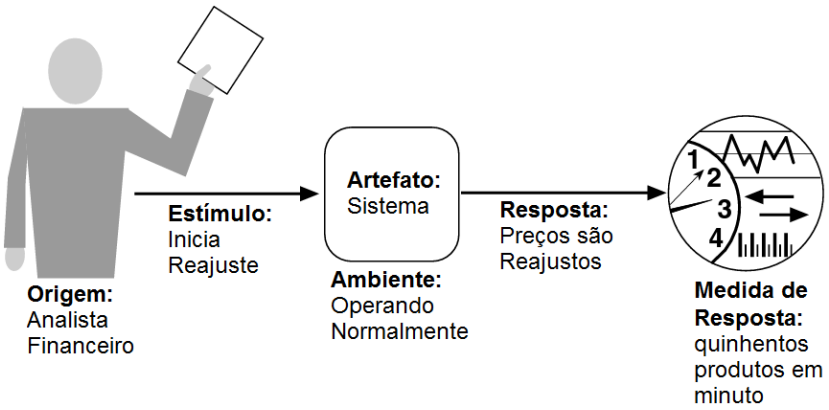
Os cenários deverão ser criados baseando-se nos fatores de Negócio previamente levantados. O objetivo principal da criação destes cenários é a obtenção dos atributos de qualidade, a fim de mostrar que a arquitetura candidata pode atendê-los. Os itens a seguir mostram como cada cenário atende aos atributos de qualidade importantes para o ERP do NUSIS.

3.3.1.1 Cenário de Desempenho

Desempenho se refere a tempo, isto é, quanto tempo o sistema demora em responder a solicitações de usuários, processamentos, entre outros (Bass, 2006). Para medir o atributo desempenho foi criado um cenário onde existe um número alto de leituras e escritas e ainda cálculos matemáticas que possam dar uma dimensão do que o software pode atingir. A Figura 31 mostra o cenário de desempenho para o protótipo de software proposto.

O Cenário descreve o analista financeiro executando uma rotina de reajuste de tabela de preço, o sistema deve ter um desempenho que permite o reajuste do preço de até 500 produtos por minuto.

Figura 31 – Cenário de Desempenho



Fonte: o próprio autor

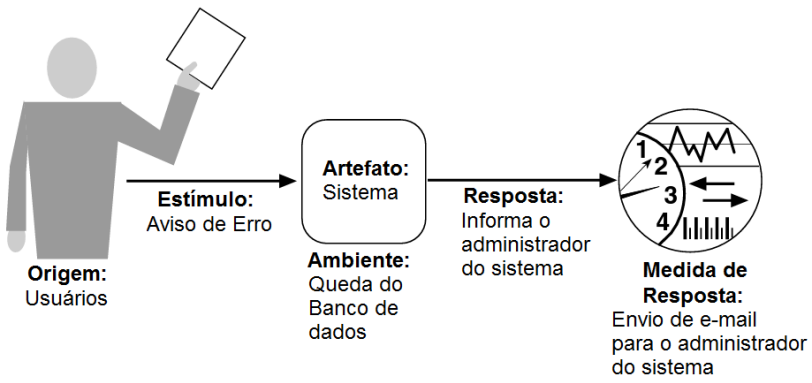
3.3.1.2 Cenário de Disponibilidade

A disponibilidade se concentra na falha do sistema e suas consequências associadas, ou seja, quando o sistema não proporciona um serviço conforme foi especificado. A recuperação ou reparação também são aspectos importantes da disponibilidade. Táticas de disponibilidade proporcionam ao usuário uma forma de mascarar a falha ou mesmo um reparo no sistema em um curto espaço de tempo (Bass, 2006). Para Bass (2006) todos os enfoques relacionados à disponibilidade envolvem algum tipo de redundância, ou um tipo de vigilância para detectar uma falha e ainda algum tipo de recuperação, uns automáticos e outros manuais (Bass, 2006).

O ERP do NUSIS será projetado para rodar em empresas de pequeno porte, seria inviável a implantação de redundância pelo seu alto custo, dessa forma optou-se pela implementação de uma vigilância. Um problema comum em empresas pequenas é a falha de rede ou até mesmo de algum recurso no servidor. A Figura 32 mostra o cenário de disponibilidade para o protótipo de software proposto.

O cenário de disponibilidade descreve que qualquer requisição efetuada por um usuário, que necessite de comunicação com o banco de dados ou até mesmo com o servidor de banco de dados e não obtiver resposta, será informado através de e-mail automático o Administrador do sistema, que uma falha aconteceu e deverá ser corrigida.

Figura 32 – Cenário de Disponibilidade

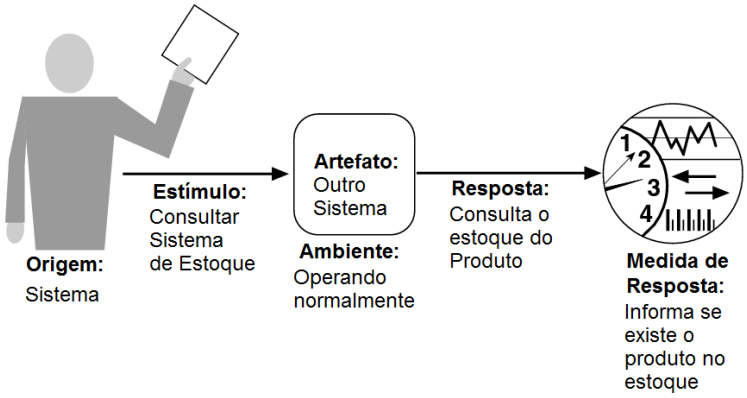


Fonte: o próprio autor

3.3.1.3 Cenário de Interoperabilidade

A interoperabilidade é a capacidade que o sistema tem de interagir com outros sistemas (Barbosa, 2009). Uma das características de um software ERP é a capacidade de se integrar com sistema legados, isto é, um software que já está implantado na empresa e que por vezes fornece informações para um nicho especifico. A fim de demonstrar que a arquitetura candidata proposta para o NUSIS atende a este atributo qualidade foi proposto o seguinte cenário de interoperabilidade. A Figura 33 mostra o cenário de interoperabilidade para o protótipo de software proposto.

Figura 33 – Cenário de Interoperabilidade



Fonte: o próprio autor

O ERP do NUSIS deve possuir a capacidade de se comunicar com um sistema de terceiros a fim de respeitar o legado. O cenário escolhido demonstra a comunicação do protótipo com um sistema de estoque terceiro, verificando se existem produtos disponíveis no estoque para a inclusão de um pedido de venda.

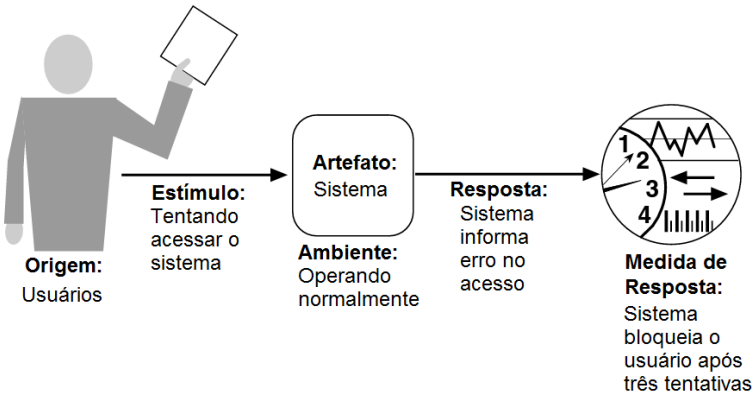
3.3.1.4 Cenário de Segurança

Segurança é a medida da capacidade do sistema de resistir à utilização não autorizada, enquanto continua a fornecer os seus serviços a usuários legítimos (Bass, 2006). Existem inúmeras formas de se implantar segurança e um *software* e cada vez isso se torna necessário. A criptografia da informação e protocolos de segurança como SSL (*Secure Socket Layer*) são

muito utilizados atualmente. Contudo em *softwares* disponíveis na WEB um ponto de fácil acesso é ainda pelo *login* do usuário, injeção SQL e quebra de senha por força bruta são técnicas muito utilizadas. Injeção SQL é quando algum invasor insere um código SQL no campo de senha para liberar o acesso, esse problema não é tratado pelo Spring Security. A outra maneira a chamada força bruta, se dá através da utilização de um software que no modo de tentativa e erro testa todas as senha possíveis para um usuário. A partir disto o cenário de segurança proposto conforme ilustra a Figura 34 foi criado.

Nesse caso o usuário ao tentar acessar o sistema tem três tentativas até que o mesmo seja bloqueado, impossibilitando que seja utilizada a força bruta nesse caso.

Figura 34 – Cenário de Segurança



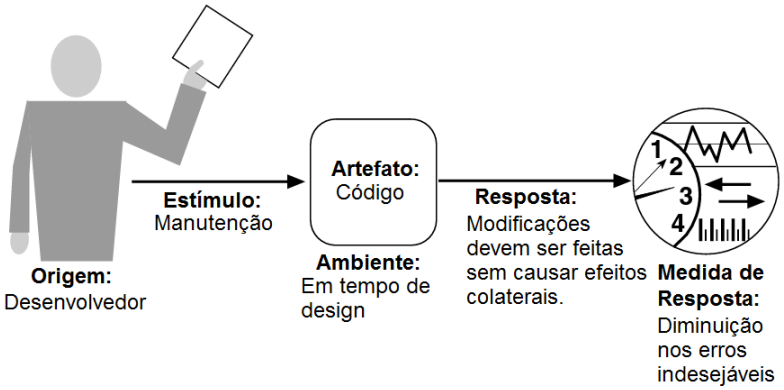
Fonte: o próprio autor

3.3.1.5 Cenário de Modificabilidade

Modificabilidade se refere ao custo da mudança (Bass, 2006). Para Barbosa (2009) quanto mais modificável é o software, menor será o impacto da mudança em áreas que teoricamente não estão relacionadas às mudanças. Segundo Pressman (2006) um WebApp é um software que exige uma evolução continuada ao contrário softwares convencionais. Desta forma convém a divisão em camadas visando a separação de interesses. Tornando o software mais fácil de ser mantido, tanto para correção de erros ou para a adição de novas funcionalidades. A Figura 35 mostra o cenário de modificabilidade para o protótipo de software proposto.

O objetivo deste cenário é que qualquer mudança executada por um desenvolvedor no software, deve afetar somente a área onde a mudança está sendo implementada.

Figura 35 – Cenário de Modificabilidade

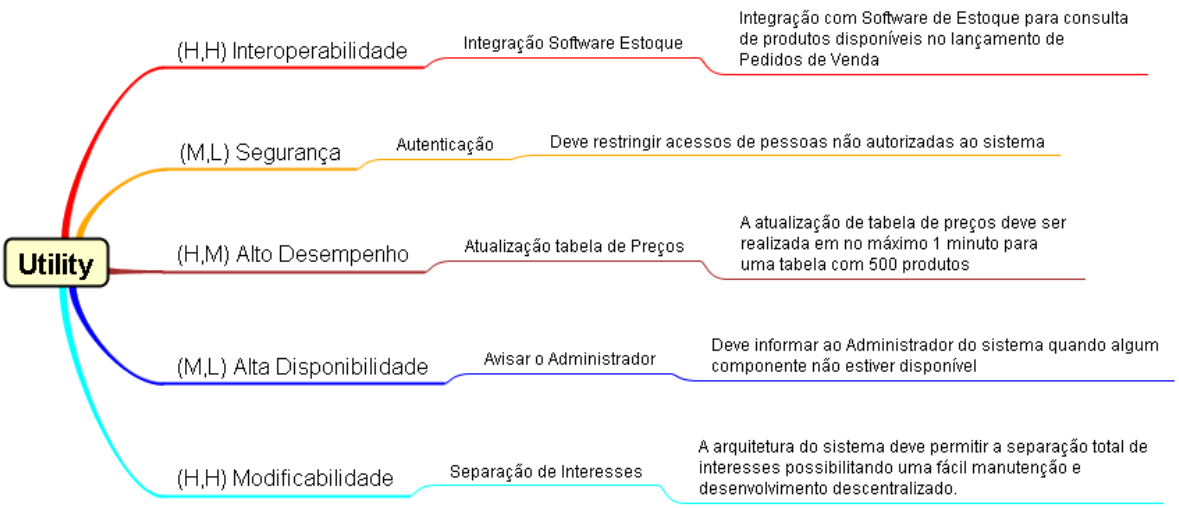


Fonte: o próprio autor

3.3.2 Árvore de Utilidade para o Protótipo do NUSIS

Os atributos de qualidade levantados através da pesquisa de softwares ERP e WebApps trazidos na etapa 2, servirão de apoio para a geração da árvore de utilidade. Que será utilizada na priorização dos cenários para o Protótipo que será desenvolvido. A Figura 36 traz a árvore de utilidade gerada a partir dos cenários levantados.

Figura 36 – Árvore de utilidade para o Protótipo do NUSIS



Fonte: O próprio Autor

3.4 CONSIDERAÇÕES FINAIS SOBRE A ARQUITETURA

O projeto do ERP do NUSIS ainda encontra-se fase de iniciação do processo. As decisões tomadas até então, tem como objetivo definir a arquitetura candidata do software a ser desenvolvido. A partir do desenvolvimento do software e do conhecimento de seus requisitos funcionais, novos atributos de qualidade surgirão. Novas decisões arquiteturais poderão ser tomadas. A arquitetura deverá evoluir juntamente com o desenvolvimento do software (Scott, 2003).

A equipe alocada no projeto será formada em sua maioria por alunos do CCTI, que serão os desenvolvedores. Uma alta rotatividade em um projeto poderia ocasionar uma codificação sem padrões tornando o entendimento e a manutenção da aplicação mais complicados. A adoção de um *framework* pode trazer além de outros benefícios, o ganho de produtividade, a padronização do código, deixando o software mais fácil de ser desenvolvido e mantido por diferentes equipes.

A escolha do Spring Framework que além de ser mantido por uma das grandes do setor de informática, a VMware, foi baseada nas suas características. É um *framework* menos intrusivo que os demais comparados, possui uma gama de alternativas em relação a segurança, integração e web que facilitam o desenvolvimento e também aumentam a produtividade da equipe (Walls, 2011). Além disso, não pretende substituir a especificação Java e sim complementa-la. O fato do Spring Framework atender as diferentes camadas da aplicação também torna mais simples o aprendizado, já que não será necessário aprender diversos *frameworks,* podendo optar por apenas um.

A definição da arquitetura teve como principal objetivo a obtenção direta dos atributos de qualidade. Bass (2006) cita questão como concorrência e processos como partes integrantes de uma arquitetura. A concorrência pode ser tratada pelo Spring Framework pelo seu módulo de Controle de Transações. Existem níveis de isolamento que podem ser configurador com no Spring, são eles: ISOLATION_DEFAULT é usado o nível de isolamento padrão do banco de dados. ISOLATION_READ_UNCOMMITTED que pode resultar em leituras sujas, não repetíveis e fantasmas. ISOLATION_READ_COMMITTED que evita leituras sujas. ISOLATION_REPEATEBLE_READ que evita leituras sujas e não repetíveis. ISOLATION_SERIALIZABLE que evita todos os problemas listados anteriormente. Em relação a estrutura de Processos que são os chamados nós de processamento ou *threads* físicas o Spring permite implementar um padrão chamado JMS

(Java Message Service) que segundo Wall (2011) é uma API (Aplication Programming Interface) que permite a troca de mensagens entre aplicações, sendo assim possível a distribuição do sistema em vários nós de processamento.

A fim de analisar se arquitetura candidata proposta atenderá aos requisitos impostos pelo *software* deverá ser aplicado o método ATAM que foi concebido pelo SEI. O ATAM é um método completo de análise arquitetural contemplando nove fases específicas e que podem auxiliar na tomada de decisão. Como será um projeto com a participação de diversas áreas é importante o envolvimento de todos em prol do *software*. A importância do ATAM está na união das pessoas envolvidas, a fim de se chegar à melhor arquitetura para o *software.*

4. PROTÓTIPO PARA SIMULAÇÃO DOS CENÁRIOS PROPOSTOS NO ATAM

No capítulo três foi definida a arquitetura e o framework que servirão de base para o desenvolvimento do ERP do NUSIS. A fim de validar se a arquitetura proposta será capaz de atender aos Fatores de Negócio elicitados, um protótipo de software foi desenvolvido.

O protótipo de software foi desenvolvido a partir do IDE Eclipse que é uma plataforma de desenvolvimento de código-fonte aberto. Segundo o próprio site do Eclipse, eclipse.org, "é uma comunidade de código-fonte aberto cujos projetos concentram-se em fornecer uma plataforma de desenvolvimento e frameworks de aplicativos baseados em código-fonte aberto". A equipe do SpringSource mantenedora do Spring Framework desenvolveu uma distribuição customizada do Eclipse chamada SpringSource Tool Suite (STS) que pode ser baixada do em http://www.springsource.org/downloads/sts. Esse ambiente de desenvolvimento já possui alguns plug-ins para o desenvolvimento através do Spring Framework.

Para simplificar o processo de compilação foi utilizado o Maven que é um aplicativo da Apache. É uma ferramenta de gestão de projetos que utiliza um arquivo XML chamado POM.xml para descrever e gerir o projeto. Em resumo o Maven baixa todos os arquivos jar´s (Executáveis do Java) necessários para seu projeto, com uma simples alteração no arquivo pom.xml, se obtém facilmente uma nova dependência, como mostra a Figura 37.

Figura 37 – Configuração das dependências no Maven

```
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-context</artifactId>
<version>3.1.1.RELEASE</version>
</dependency>
```

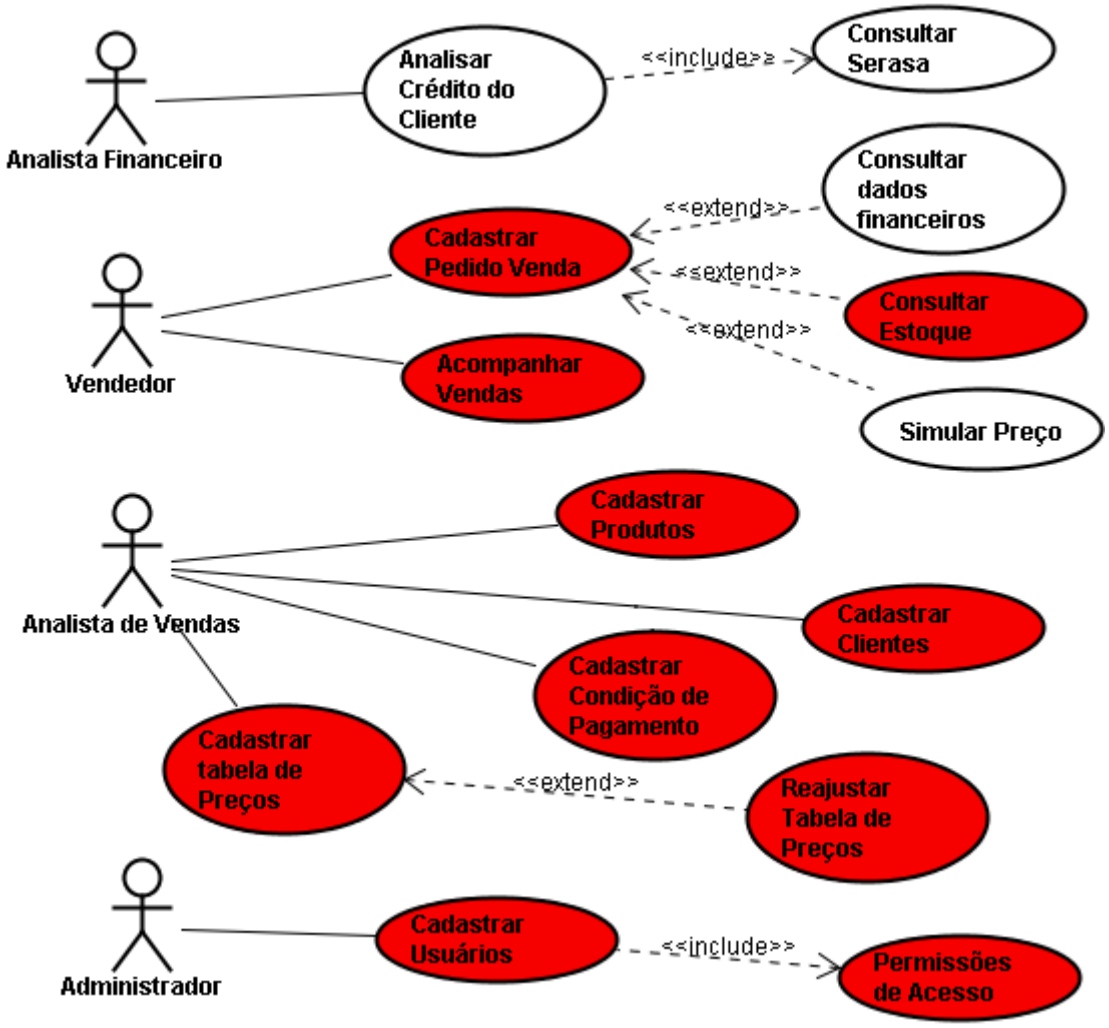
Fonte: o próprio autor

Com isso se torna mais fácil compartilhar *plugins* por todos os projetos proporcionando um sistema de construção uniforme.

O protótipo tem exclusivamente o objetivo de atender aos atributos de qualidade e servir como exemplo para o início de desenvolvimento do ERP WEB do NUSIS, isto é, não tem a pretensão de ser um software pronto para ser utilizado. Será um sistema de lançamento de pedidos no qual será possível visualizar se cada atributo de qualidade está sendo atendido.

O Diagrama de casos de uso foi definido e gerado juntamente com os interessados no projeto, os casos de uso que serão atendidos no protótipo estão com o balão em vermelho como ilustra a Figura 38.

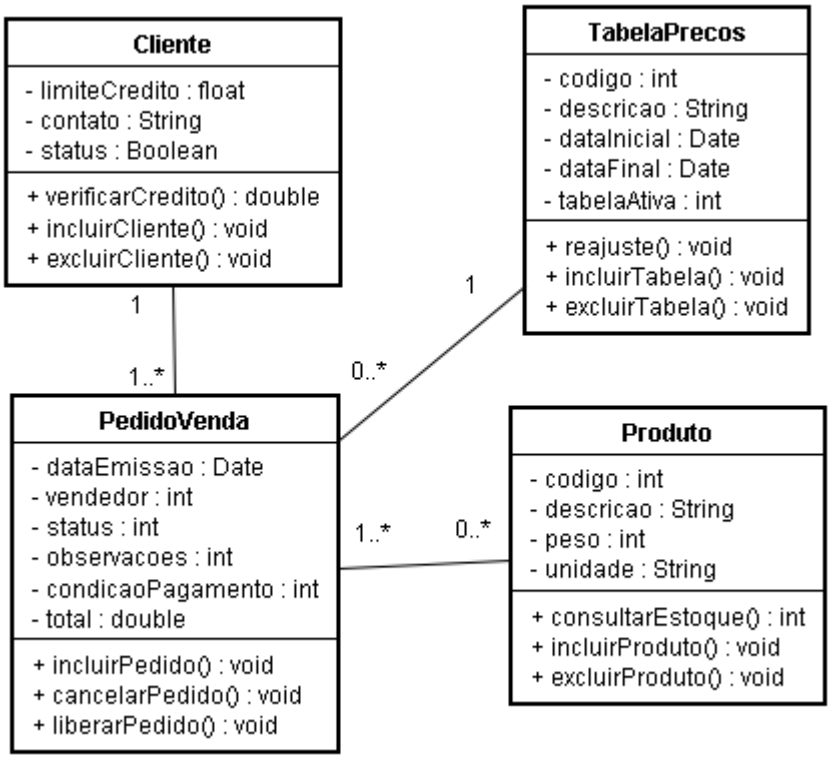
Figura 38 – Casos de Uso do Protótipo



Fonte: o próprio autor

O Diagrama de Classes ilustrado na Figura 39 mostra a decomposição do software em objetos o que torna mais fácil sua compreensão e comparação com o mundo real.

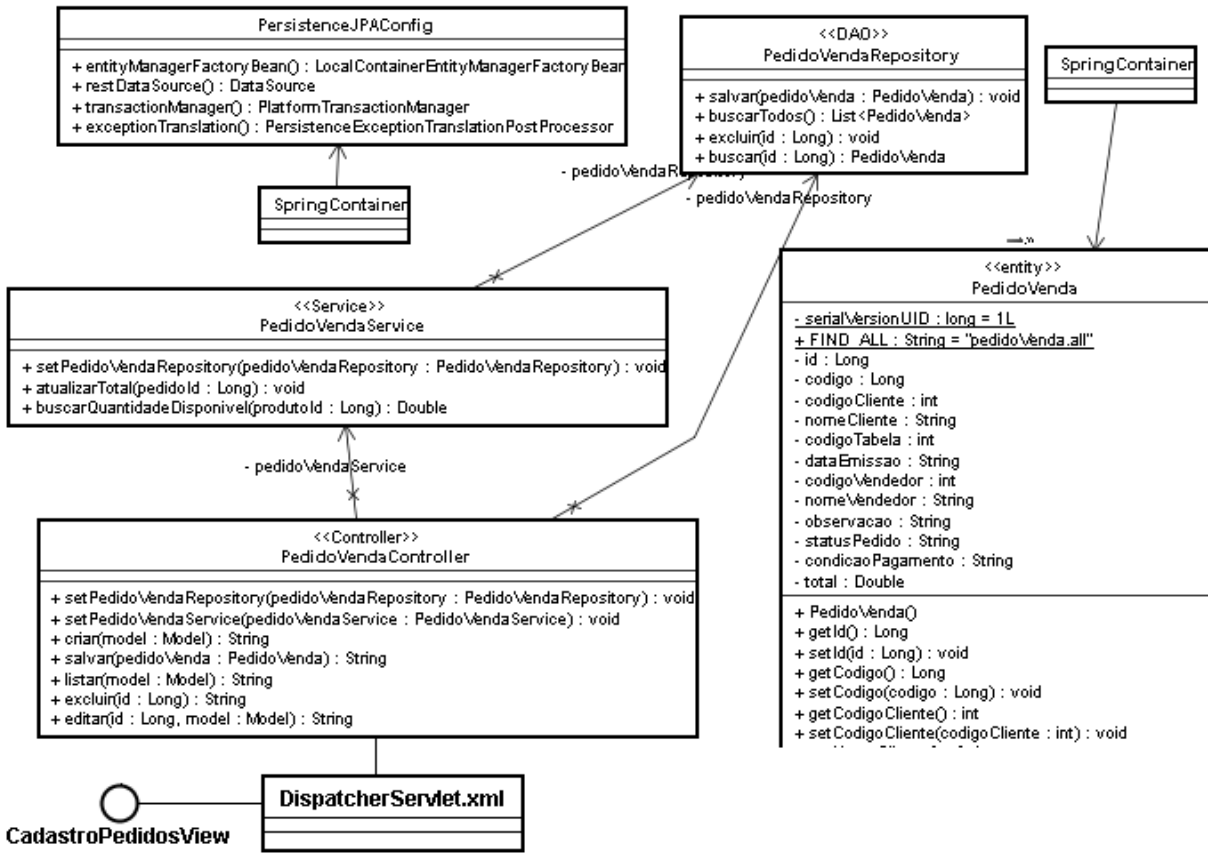
Figura 39 – Diagrama de Classes de Análise para o protótipo



Fonte: o próprio autor

O desenvolvimento para o ERP do NUSIS deve seguir o modelo conforme a Figura 40.

Figura 40 – Modelo de diagrama de classes para o NUSIS



Fonte: o próprio autor

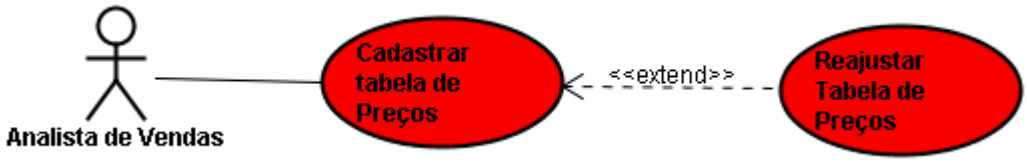
A Figura 40 trouxe parte da rotina de cadastro de pedidos de venda. Qualquer requisição realizada pelo usuário chega ao arquivo DispatcherServlet.xml utilizado pelo Spring Framework para encontrar as classes *Controller* que por sua vez tem a função de ser um controlador, responsável por interpretar as ações do usuário e as mapeia para as chamadas do modelo. As entidades são instanciadas pelo Spring Container, dessa forma não é criada uma dependência direta com elas, deixando a cargo do container de inversão de controle a tarefa de cria-las.

O protótipo de *software* deverá atender aos cinco atributos de qualidade levantados. São eles: Modificabilidade, Segurança, Desempenho, Disponibilidade e Interoperabilidade. Cada um desses atributos de qualidade será melhor detalhado a seguir.

4.1 DESEMPENHO

O atributo de qualidade é representado pelo caso de uso “Cadastrar Tabela de Preços” onde será possível através da implementação da funcionalidade “Reajustar Tabela de Preços” a alteração de todos os produtos cadastrados nela. A Figura 41 representa o caso de uso em questão.

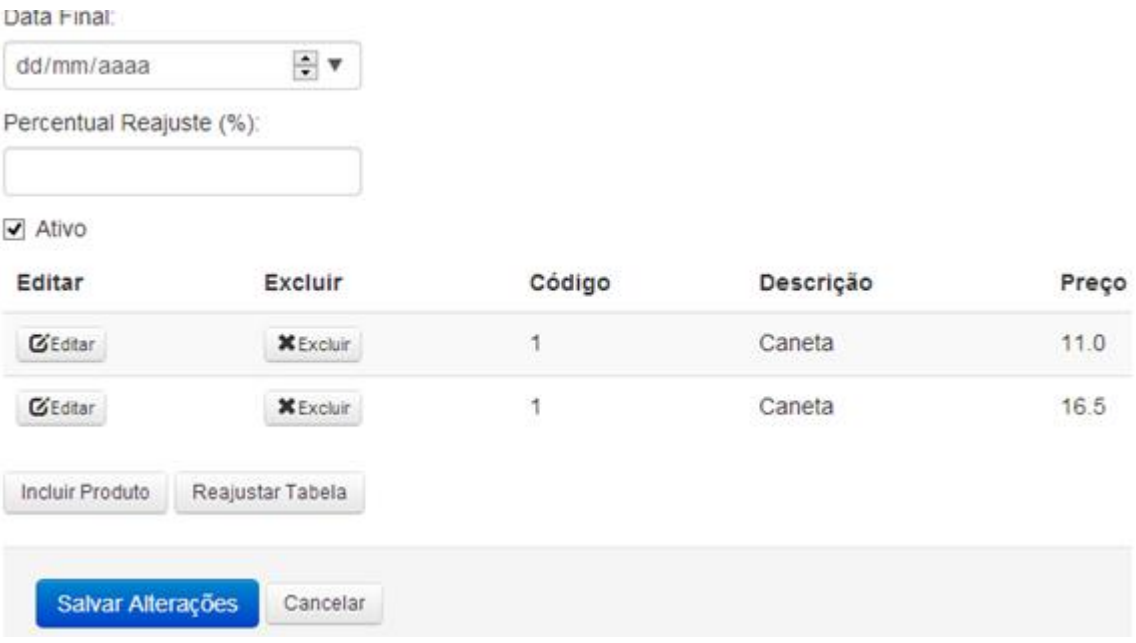
Figura 41 – Caso de Uso Cadastrar Tabela de Preços



Fonte: o próprio autor

A Figura 42 demonstra o formulário de cadastro de tabela de preços onde existe o campo para inserção do percentual de reajuste, e o botão onde é aplicado o reajuste de preço. O sistema deverá pesquisar os preços dos itens cadastrados na tabela de preço e multiplicar pelo percentual de reajuste inserido pelo usuário. Foi realizado o teste que permitiu o ajuste de uma tabela de preços contendo 5000 itens cadastrados em apenas 5 segundos.

Figura 42 – Formulário com a funcionalidade para Reajuste de Preços



Fonte: o próprio autor

4.2 INTEROPERABILIDADE

Para demonstrar que a arquitetura proposta atende a esse atributo de qualidade o sistema de pedidos de venda foi integrado a um sistema de estoque de terceiro.

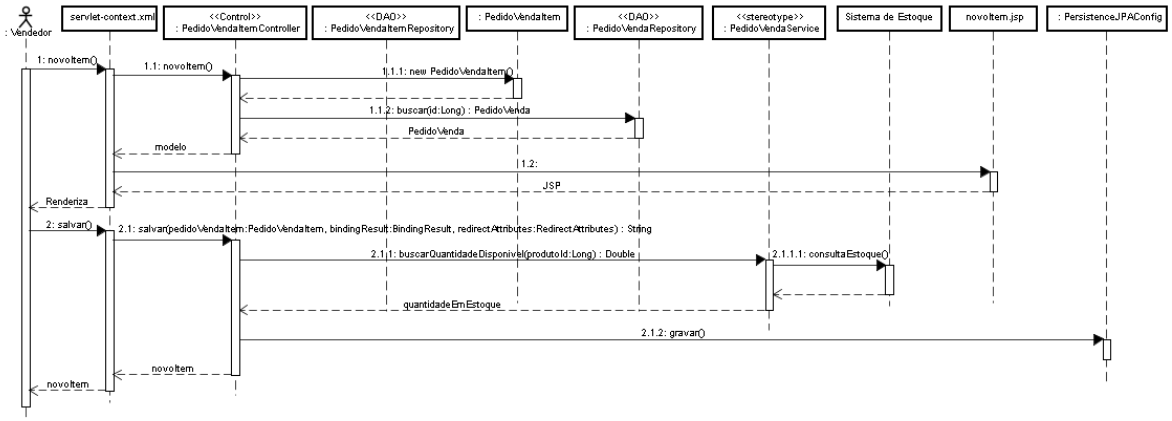
Esta funcionalidade é demonstrada através do caso de uso Cadastrar Pedido de Venda (Figura 43).

Figura 43 – Caso de Uso Cadastrar Pedido Venda

Fonte – o próprio autor

A fim de demonstrar a sequência de processos da funcionalidade foi desenvolvido o diagrama de sequência que é demonstrado na Figura 44. Podendo ser melhor visualizada no Apendice A.

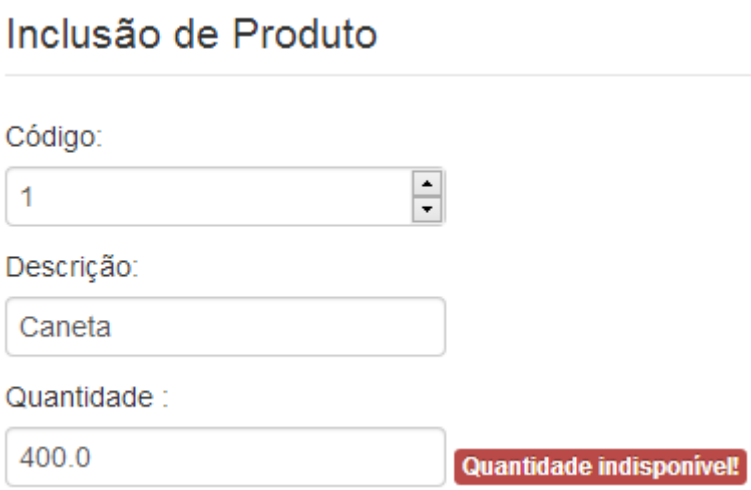
Figura 44 – Diagrama de Sequência para Interoperabilidade



Fonte: o próprio autor

O Vendedor ao incluir um produto no pedido de venda, o sistema de pedidos consulta no sistema de estoque a quantidade do produto que está disponível em estoque, caso não esteja disponível é demonstrada a mensagem Quantidade Indisponível, como é mostrado na Figura 45. A integração foi realizada utilizando um *template* do próprio Spring framework chamado RestTemplate utilizando para a troca de mensagem o é disponibilizado um endereço HTTP e a resposta é armazenada em um DTO (*Data Transfer Object*).

Figura 45 – Integração Sistema de Pedidos com Sistema de Estoque



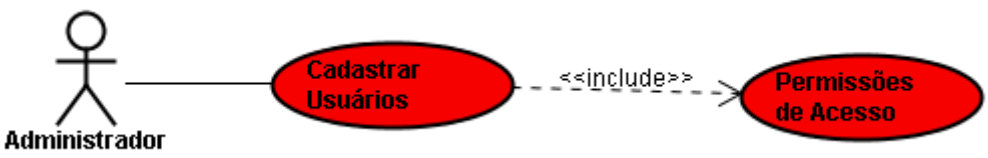
Fonte – o próprio autor

Para realizar o teste foi utilizado o cenário de interoperabilidade. Conforme ilustrou a Figura 45 foi inserida a quantidade 400 para o item Caneta, retornando a mensagem "Quantidade indisponível", pois não possuía essa quantidade em estoque, após realizar a consulta no sistema de estoque terceiro.

4.3 SEGURANÇA

O caso de uso representado na Figura 46 visa atender o cadastro de usuário bem como, permissões de acesso, isto é, autenticação e autorização, duas das premissas para o atributo de qualidade segurança, que são plenamente atendidos pelo framework Spring Security.

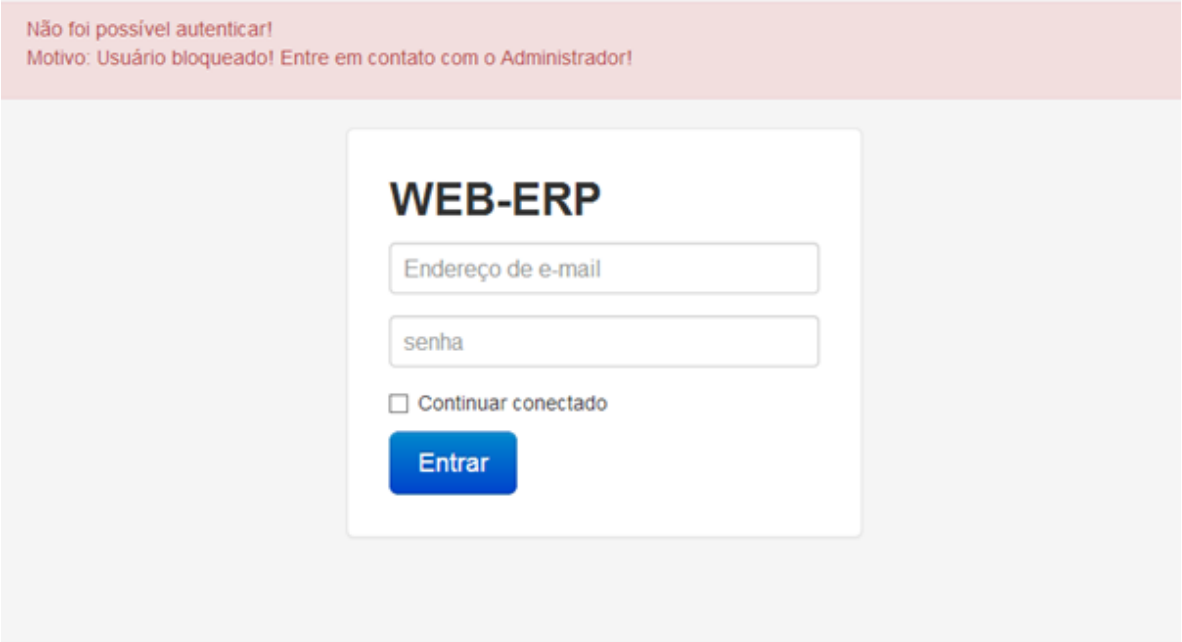
Figura 46 – Caso de uso Cadastrar Usuários



Fonte: o próprio autor

Visando aumentar a segurança de acesso foi implementado o bloqueio do usuário no caso da inserção da senha incorreta por três vezes, impossibilitando que a partir de um software de força bruta a senha seja descoberta e bloqueando o acesso indevido ao software. A Figura 47 mostra a interface de login para acesso do usuário, conforme a imagem é mostrada a mensagem “Usuário Bloqueado” informando que o mesmo encontra-se bloqueado, pelo erro de digitação ou até mesmo uma tentativa de invasão do sistema.

Figura 47 – Interface para Login do usuário

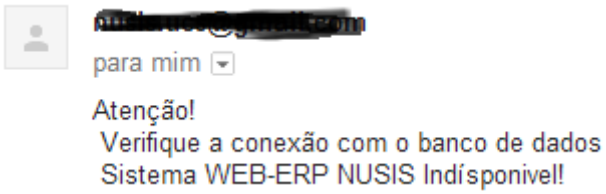


Fonte: o próprio autor

4.4 DISPONIBILIDADE

A implementação de um serviço para alta disponibilidade demandaria um alto investimento. Como disponibilidade é um atributo de qualidade que necessita ser atendido para o ERP do NUSIS e pelo foco do Projeto Software nas Empresas ser pequenas empresas se tornaria inviável implementar tal serviço. Dessa forma para o protótipo foi implementado um recurso em que a partir da queda do banco de dados é enviado um e-mail para o administrador do sistema informando do problema a Figura 48 demonstra o e-mail enviado.

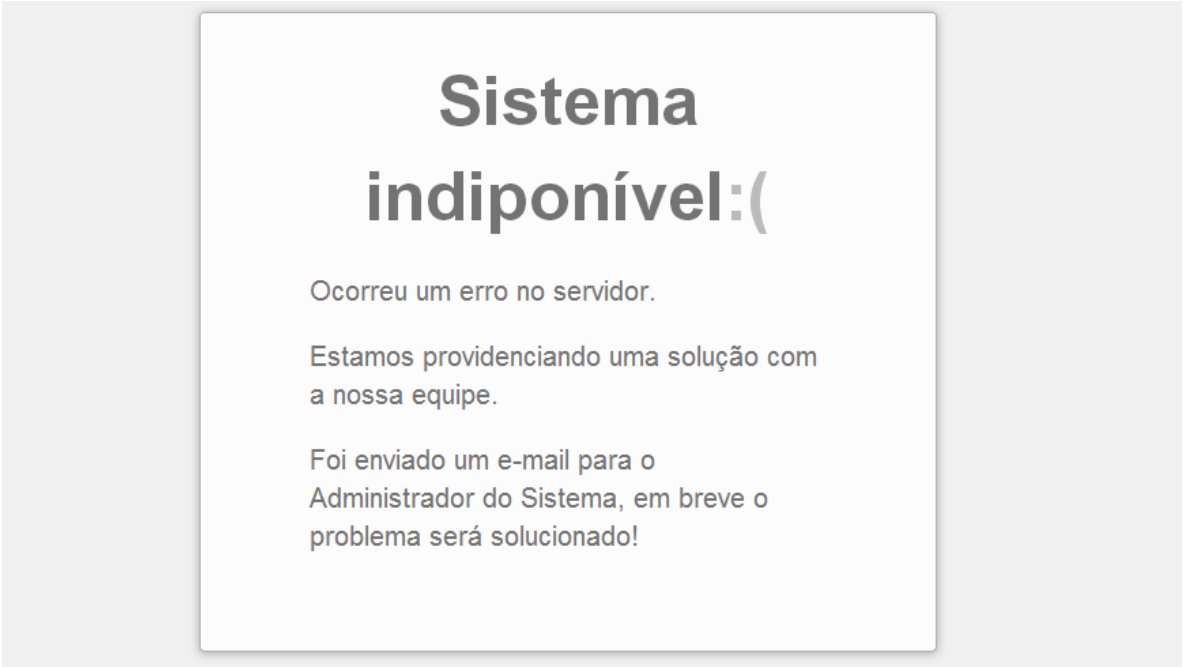
Figura 48 – E-mail enviado após queda do Banco de Dados



Fonte: o próprio autor

Também será exibida a página conforme ilustra a Figura 49 informando ao usuário que o problema estará sendo tratado pela equipe de suporte.

Figura 49 – Página de erro Sistema indisponível



Fonte: o próprio autor

4.5 MODIFICABILIDADE

Conforme Pressman (2006) um WebApp é constantemente modificado tanto para inserção de novas funcionalidades quanto para manutenção das já existentes pela correção de inconsistências ou pela mudança nos processos. A opção pela utilização de um framework de desenvolvimento orienta o desenvolvedor pela adoção de conceitos e de padrões.

O conceito MVC é amplamente utilizado em projeto de softwares baseados na WEB, dessa forma optou-se por utilizar o Framework Spring Web MVC onde se tem uma completa separação de interesses, uma premissa para a modificabilidade.

A Figura 50 demonstra uma parte do código de uma classe Controle. Essa classe é responsável por receber todas as requisições que vem do DispatcherServlet (arquivo de configuração do Spring que recebe todas as conexões dos usuários), obter os modelos e retornar aos usuários as visões. A anotação @Controller é utilizada para que o DispatcherServlet a encontre no momento de uma requisição.

Figura 50 – Código fonte do Controlador

```
@Controller
@RequestMapping("/pedidoVenda")
public class PedidoVendaController {

    private PedidoVendaRepository pedidoVendaRepository;

    private PedidoVendaService pedidoVendaService;

    @Autowired
    public void setPedidoVendaRepository(PedidoVendaRepository pedidoVendaRepository) {
        this.pedidoVendaRepository = pedidoVendaRepository;
    }

    @Autowired
    public void setPedidoVendaService(PedidoVendaService pedidoVendaService) {
        this.pedidoVendaService = pedidoVendaService;
    }

    @RequestMapping(value = "/novo", method = RequestMethod.GET)
    public String criar(Model model) {
        model.addAttribute(new PedidoVenda());
        return "pedidoVenda/editar";
    }
```

Fonte: o próprio autor

A Figura 51 mostra parte do código de uma classe Repositório, é a classe responsável pela manipulação e persistências dos objetos. A anotação @Repository deve ser utilizada para que o Spring entenda que essa classe se trata de um DAO (*Data Access Objet*).

Figura 51 – Código fonte da classe de persistência

```
@Repository
@Transactional
public class PedidoVendaRepository {

    @PersistenceContext
    private EntityManager em;

    public void salvar(PedidoVenda pedidoVenda) {
        if (pedidoVenda.getId() == null) {
            em.persist(pedidoVenda);
        } else {
            PedidoVenda pedidoVendaGravado = em.find(PedidoVenda.class, pedidoVenda.getId());
            pedidoVenda.setCodigo(pedidoVendaGravado.getCodigo());
            em.merge(pedidoVenda);
        }
    }
```

Fonte: o próprio autor

A Figura 52 mostra uma classe POJO que nada mais é que um mapeamento do objeto que será persistido no banco de dados. Esta classe é anotada com a anotação @Entity, informando ao Spring que se trata de uma entidade.

Figura 52 – Entidade mapeamento do Objeto Relacional

```
@Entity
@Table(name = "pedido_venda")
@NamedQueries(value = {@NamedQuery(name=PedidoVenda.FIND_ALL, query = "FROM PedidoVenda")})
public class PedidoVenda implements Serializable {

    private static final long serialVersionUID = 1L;

    public static final String FIND_ALL = "pedidoVenda.all";

    @Id
    @Column(name = "id", nullable = false)
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "codigo", nullable = false)
    private Long codigo;
```

Fonte: o próprio autor

A visão é que um arquivo JSP em linguagem HTML. O Spring Framework possui bibliotecas que facilitam o desenvolvimento. Com diversos componentes prontos e algumas validações ele aumenta a produtividade para o desenvolvimento da interface com o usuário. A Figura 53 traz um trecho do código de um arquivo JSP.

Figura 53 – Código fonte da Interface com o usuário

```
</div>
<div class="row">
    <div class="span6">
        <div class="row-fluid">
            <div class="span6">
                <form:label path="codigoCliente"><spring:message code="label.PVcodCli"/>:</form:label>
                <form:input path="codigoCliente" type="number" id="input-codCli" onChange="buscarNomeCli();"/>
            </div>
            <div>
                <form:label path="nomeCliente"><spring:message code="label.PVnomeCli"/>:</form:label>
                <form:input path="nomeCliente" class="input-xlarge" readonly="true" id="input-nomeCli"/>
            </div>
        </div>
    </div>
</div>
```

Fonte: o próprio autor

4.6 CONSIDERAÇÕES FINAIS

O ATAM não possui uma etapa para prototipação de *software*. Com o intuito de comprovar se as escolhas que resultaram na arquitetura candidata podem atender aos requisitos do ERP do NUSIS foi implementado um protótipo de *software.*

O desenvolvimento do protótipo teve única e exclusivamente o objetivo de atender aos atributos de qualidade que foram apresentados na etapa que descrevia os principais fatores de negócio da aplicação. Cada atributo de qualidade levantado foi atendido pelo protótipo e provado através de testes explicados durante o capítulo.

A arquitetura candidata será realmente testada e afirmada somente no momento em que for aplicado o método ATAM em situações reais, envolvendo todos os membros da equipe e também quando o ERP estiver sendo implementado. Caso a arquitetura candidata proposta não atenda às características impostas pelo ERP do NUSIS está poderá ser refinada, pois se trata de um processo iterativo, passando novamente pelo método da análise, a fim de se chegar a qualidade desejada.

5. CONCLUSÃO

O desenvolvimento deste trabalho permitiu elicitar requisitos necessários para a definição de uma arquitetura candidata para o projeto do NUSIS na fase de iniciação, através de problemas conhecidos ou experiências passadas, com a utilização de métodos, estruturas e padrões arquiteturais existentes. Colocando em prática através de implementação de um protótipo de software visando atender aos requisitos impostos.

Primeiramente foi apresentado o estudo bibliográfico sobre as estruturas, com os seus devidos componentes e conectores que juntos formarão a arquitetura de *software*. Incluindo o estudo sobre frameworks, que já implementam algumas dessas decisões de projeto, contribuindo para o aumento da produtividade deixando ao desenvolvedor somente as decisões especificas da aplicação que está sendo desenvolvida e também a padronização de código, já que obrigam de certa forma o desenvolvedor a implementar a partir de uma estrutura similar. E por fim, como saber se a arquitetura proposta é correta? Não existe uma resposta 100% verdadeira, existem sim métodos que diminuem os riscos destas escolhas, foram apresentados três métodos, o SAAM, o ATAM e o CBAM, este último mais focado no custo, enquanto os outros dois mais focados na definição de uma arquitetura baseando-se nos atributos de qualidade exigidos. Esses métodos se aplicados em fases iniciais no processo de desenvolvimento tornam as mudanças menos custosas, pois a arquitetura ainda não foi implementada. Estes estudos contribuíram para a obtenção de conhecimento das diferentes estruturas, padrões, projeto e métodos de análise arquitetural. Permitindo embasar as escolhas necessárias para o ERP do projeto do NUSIS.

O capítulo 3 trouxe a definição do método de análise arquitetural ATAM. Foi optado pelo ATAM devido a complexidade do projeto, bem como a característica da equipe por ser formada por alunos. Uma equipe grande inexperiente e com uma alta rotatividade, necessita de um envolvimento maior por parte do corpo docente. O ATAM traz fases em que é envolvida uma equipe de avaliação, gerando debates e melhorando a comunicação entre todos os envolvidos, a fim de se chegar a um consenso em relação à arquitetura. Nas primeiras fases do ATAM as definições são provenientes de um arquiteto de *software* que toma as decisões, nas fases seguintes podem ser apreciadas pela equipe de avaliação, para adequações caso necessário.

Dentre as escolhas realizadas foi definido que o software será desenvolvimento com a utilização de um framework. Foram citados os frameworks Struts da Apache que foi um dos

primeiros frameworks web desenvolvidos, o JSF que é um framework MVC da própria especificação Java e o Spring Framework mantido pela VMware. Foi optado pela utilização do Spring primeiramente por ser leve, ou seja, é pouco intrusivo, tornando mais fácil a troca por outro, caso o NUSIS não venha a adotar esta ferramenta. Outro motivo é que o Spring possui uma gama de componentes, permitindo adota-lo em todas as camadas da aplicação.

O método ATAM não possui uma etapa focada para a prototipação dos cenários em um *software.* Sendo assim para gerar a primeira iteração no processo de desenvolvimento da arquitetura, também visando servir como base para o início do processo de desenvolvimento do ERP do NUSIS, foi desenvolvido um protótipo de *software*. O *software* tem o objetivo de atender aos atributos de qualidade elicitados e foi desenvolvido na linguagem Java utilizando o Spring Framework como base.

Como foi realizada apenas uma simulação do método de análise arquitetural, e o software desenvolvido é apenas um protótipo, não será possível saber se a arquitetura de software candidata poderá aderir aos requisitos do ERP do NUSIS. Além disso, a arquitetura somente será realmente avaliada quando o *software* estiver pronto e implantado em produção, contudo servirá como ponto de partida.

Em busca de melhor aprimoramento sobre a definição da arquitetura do *software*, é importante ter em mente que o seu desenvolvimento é um processo iterativo e pode ser refinado ao longo do projeto do NUSIS.

REFERÊNCIAS BIBLIOGRÁFICAS

BARRETO, Celso. **Agregando Frameworks de Infra-Estrutura em uma Arquitetura Baseada em Componentes: Um Estudo de Caso no Ambiente AulaNet.** Rio de Janeiro. 2006.

BASS, Len. ***Software* Architecture in Practice.** 2º ed, Addison Wesley. 2006.

BOOCH, Grady. **UML:** guia do usuário**.** Rio de Janeiro 6ºed. Editora Elsevies. 2005.

CAIÇARA Júnior, Cícero. **Sistemas Integrados de Gestão ERP:** Uma abordagem Gerencial. 3º ed. Ibpex. 2008.

COULOURIS, George. **Sistemas Distribuidos:** conceitos e projeto. 4º ed. Bookman., 2007.

FOWLER, Martim. **Padrões de Arquitetura de Aplicações Corporativas.** 2006. 40 78-92. 315p.

GAMA, Erich. **Padrões de Projeto:** soluções reutilizáveis de software orientado a objetos**.** Bookman. 2000.

GODERIS, Sofie. **On the Separation of User Interface Concerns:** A Programmer´s Perspective on the Modularisation of User Interface Code. 2008. VUBPRESS.

HEMRAJANI, Anil. **Desenvolvimento Ágil em Java com Spring, Hibernate e Eclipse**, Pearson Prentice Hall. 2007.

JSON. <json.org> acesso em 1 jun 2013.

KEITH, Mike. Pro JPA 2 Mastering the Java Persistence API. Apress. 2009. p 33-40.

KITO, Mann. Java Server Faces in Action. Manning. 2005.

LARMAN, Craig. **Utilizando UML e Padrões.** 3º ed. Bookman. 2007. 219-225p.

MASIERO, Paulo **Padrões e Frameworks de *Software***
<http://www2.icmc.usp.br/~rtvb/apostila.pdf> acesso em 28/11/2012

MENDES, Antonio – **Arquitetura de *Software* – Desenvolvimento orientado a arquitetura**, 2002. 16,20,132p

NBR ISO/IEC 9126 – **Engenharia de *Software*** – Qualidade de Produto, 2003.

PRESSMAN Roger – **Engenharia de *Software***, 6ª ed, 2006. 378-381p

SCOTT Kendall – **O Processo Unificado Explicado**. Addison-Wesley, 2003.

SOMMERVILE, Ian – **Engenharia de *Software***, 9ª ed. Pearson Prentice Hall, 2011. 104-218p

Spring Source <http://static.springsource.org/spring-integration/reference/pdf/spring-integration-reference.pdf> acesso em: 30 abr. de 2013.

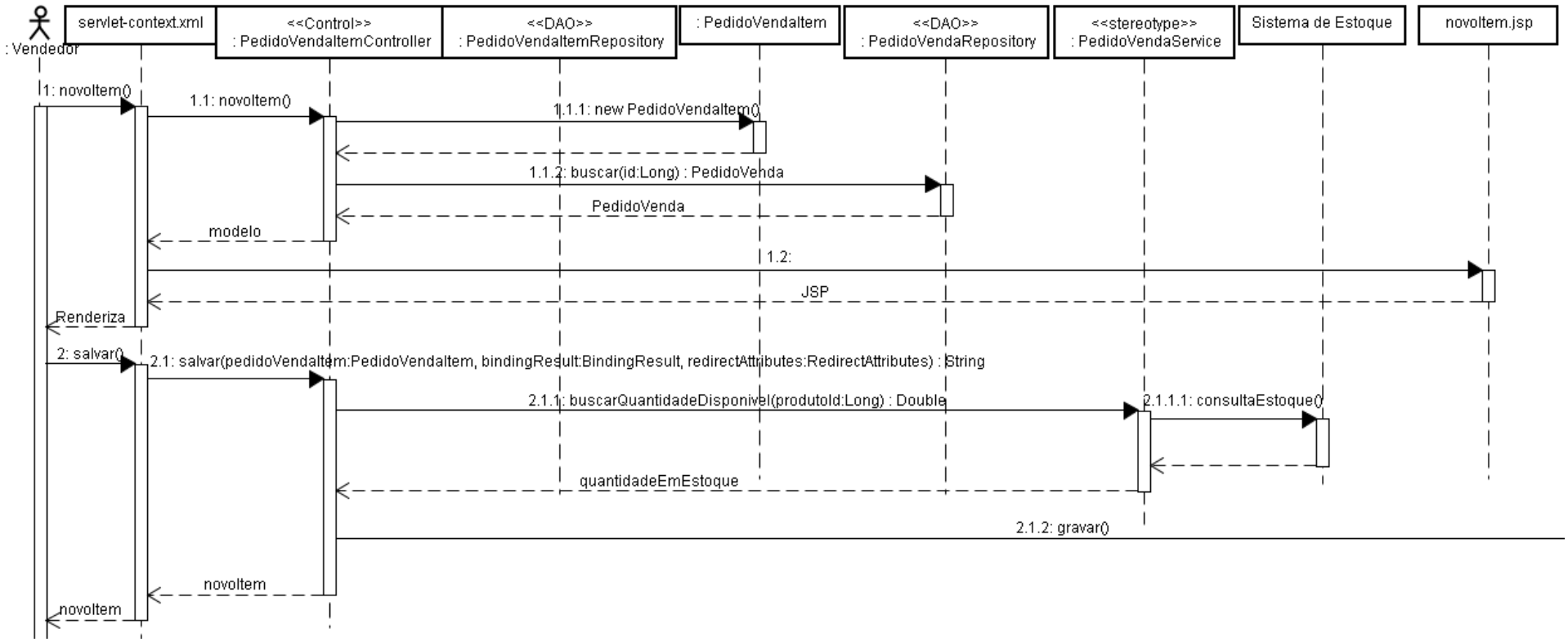
Rational Unified Process <http://www.wthreex.com/rup/process/modguide/md_impid.htm> acesso em 30/10/2012.

The Architecture Tradeoff Analysis Method
<http://www.sei.cmu.edu/architecture/tools/evaluate/atam.cfm> acesso em 03 dez. 2012>

WALLS, Craig – **Spring in Action,** 3º ed, Shelter Island, 2011.

WEINSSMANN, Henrique – **Vire o jogo com Spring Framework**, Casa do Código, 2012.

APÊNDICE A


: Vendedor
servlet-context.xml
<<Control>>
: PedidoVendaItemController
<<DAO>>
: PedidoVendaItemRepository
: PedidoVendaItem
<<DAO>>
: PedidoVendaRepository
<<stereotype>>
: PedidoVendaService
Sistema de Estoque
novoItem.jsp
1: novoItem()
1.1: novoItem()
1.1.1: new PedidoVendaItem()
1.1.2: buscar(id:Long) : PedidoVenda
PedidoVenda
modelo
1.2:
JSP
Renderiza
2: salvar()
2.1: salvar(pedidoVendaItem:PedidoVendaItem, bindingResult:BindingResult, redirectAttributes:RedirectAttributes) : String
2.1.1: buscarQuantidadeDisponivel(produtoId:Long) : Double
2.1.1.1: consultaEstoque()
quantidadeEmEstoque
2.1.2: gravar()
novoItem
novoItem