

UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ – UTFPR
CURSO SUPERIOR DE TECNOLOGIA EM DESENVOLVIMENTO DE SISTEMAS
DE INFORMAÇÃO

RADUAN SILVA DOS SANTOS

**DESENVOLVIMENTO DE APLICAÇÃO PARA O CONTROLE DE SCRUM COM
MONGODB**

TRABALHO DE DIPLOMAÇÃO

MEDIANEIRA

2011

RADUAN SILVA DOS SANTOS

**DESENVOLVIMENTO DE APLICAÇÃO PARA O CONTROLE DE SCRUM COM
MONGODB**

Trabalho de Diplomação apresentado à disciplina de Trabalho de Diplomação, do Curso Superior de Tecnologia em Desenvolvimento de Sistemas de informação da Universidade Tecnológica Federal do Paraná – UTFPR, como requisito parcial para obtenção do título de Tecnólogo.

Orientador: Prof. Me. Fernando Schütz.

MEDIANEIRA

2011



Ministério da Educação
Universidade Tecnológica Federal do Paraná
Diretoria de Graduação e Educação Profissional
Curso Superior de Tecnologia em Análise e
Desenvolvimento de Sistemas



TERMO DE APROVAÇÃO

DESENVOLVIMENTO DE APLICAÇÃO PARA O CONTROLE DE SCRUM COM MONGODB

Por

Raduan silva dos santos

Este Trabalho de Diplomação (TD) foi apresentado às 09:10 h do dia 23 de novembro de 2011 como requisito parcial para a obtenção do título de Tecnólogo no Curso Superior de Tecnologia em Análise e Desenvolvimento de sistemas, da Universidade Tecnológica Federal do Paraná, *Campus* Medianeira. Os acadêmicos foram argüidos pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado com louvor e mérito.

Prof. Fernando Schütz, Me.
UTFPR – *Campus* Medianeira
(Orientador)

Prof. Alan Gavioli, Me.
UTFPR – *Campus* Medianeira
(Convidado)

Prof. Alessandra Garbelotti Bortoletto
Hoffmann, Me.
UTFPR – *Campus* Medianeira
(Convidado)

Prof. Juliano Rodrigo Lamb, M. Eng.
UTFPR – *Campus* Medianeira
(Responsável pelas atividades de TCC)

DEDICATÓRIA

Dedicado a minha mãe, Ivana Silva dos Santos.

“Você pode encarar um erro como uma besteira a ser esquecida, ou como um resultado que aponta uma nova direção”.

Steve Jobs.

RESUMO

Santos, Raduan Silva dos. Desenvolvimento de aplicação para o controle de Scrum com MongoDB. 2011. Trabalho de conclusão de curso (Tecnologia em Desenvolvimento de Sistemas de informação), Universidade Tecnológica Federal do Paraná. Medianeira 2011.

Os bancos de dados não relacionais atualmente têm ganhado vários adeptos, devido à escalabilidade e facilidade de uso proporcionado pela grande maioria dos bancos de dados não relacionais. Paralelamente a isto, na área de gerenciamento de projetos, a metodologia ágil Scrum vem se destacando, devido à facilidade e controle do tempo gasto mais perto possíveis do real. Este trabalho apresenta um estudo sobre o Scrum e sobre a implementação de uma aplicação para gerenciamento de etapas do Scrum, visando a apresentação da implementação do banco de dados *MongoDB*, com utilização de alguns *frameworks*, como o Google Guice e o *Morphia*.

Palavras chaves: SCRUM, Gerenciamento de Projetos, MongoDB, Google Guice, Morphia.

ABSTRACT

Santos, Raduan Silva dos. Desenvolvimento de aplicação para o controle de Scrum com MongoDB. 2011. Trabalho de conclusão de curso (Tecnologia em Desenvolvimento de Sistemas de informação), Universidade Tecnológica Federal do Paraná. Medianeira 2011.

Currently the Non-relational databases have won many fans, due the scalability and ease of use, provided by the vast majority of non-relational databases. Parallel to this, in the project management area, the agile methodology Scrum has been highlighted due the ease and time-controlling, with values closer to the real.

This paper aims to present a study about Scrum and the implementation of an application for management of the Scrum steps, aimed at presenting the implementation of the MongoDB database, using some frameworks, such as Google Guice and Morphia.

Keywords: SCRUM, Gerenciamento de projetos, MongoDB, Google Guice, Morphia.

LISTA DE SIGLAS

| | |
|-------|---|
| ACID | Atomicidade,Consistência,Isolamento e Durabilidade |
| BASE | <i>Basic Availability, Soft-state, Eventual consistency</i> |
| DAO | <i>Data Access Object</i> |
| DTO | <i>Data Transfer Object</i> |
| GPA | Gerenciador de Projetos Ágeis |
| GWT | <i>Google Web Toolkit</i> |
| JAR | <i>Java Archive</i> |
| JSON | <i>Javascript Object Notation</i> |
| MER | Modelo Entidade Relacionamento |
| MS | Milissegundos |
| NOSQL | <i>Not Only SQL</i> |
| POJO | <i>Plain Old Java Objects</i> |
| VO | <i>Value Object</i> |
| XML | <i>Extensible Markup Language</i> |

LISTA DE FIGURAS

| | |
|--|----|
| Figura 1 - Ciclo do Scrum..... | 15 |
| Figura 2 - Exemplo de quadro de Scrum..... | 16 |
| Figura 3 - Criação de um banco de dados no <i>MongoDB</i> | 20 |
| Figura 4 - Inserção de dados no banco de dados selecionado | 21 |
| Figura 5 - Busca com o método <i>Find</i> | 21 |
| Figura 6 - Busca com <i>Query</i> | 22 |
| Figura 7 - <i>Query</i> com documento explícito..... | 23 |
| Figura 8 - MER da aplicação..... | 31 |
| Figura 9 - <i>Value Object</i> de produto..... | 40 |
| Figura 10 - Produto Pojo | 41 |
| Figura 11 - Objeto Transiente para conversão de <i>POJO</i> para <i>Value Object</i> | 42 |
| Figura 12 - método de conversão de <i>POJO</i> para <i>Value Object</i> | 43 |
| Figura 13- Diagrama de funcionamento da conversão de <i>POJO</i> e <i>Value Object</i> | 43 |
| Figura 14 - Constantes da classe <i>MongoConnection</i> | 44 |
| Figura 15 - Objetos estáticos utilizados nos métodos <i>singleton</i> | 45 |
| Figura 16 - Método <i>getMongo</i> | 45 |
| Figura 17 - Método <i>getMorphia</i> | 46 |
| Figura 18 - Método <i>getDatastore</i> | 47 |
| Figura 19 - Interface <i>IDao</i> | 49 |
| Figura 20 - Interface <i>IDaoProjeto</i> | 50 |
| Figura 21 - <i>DaoProjeto</i> | 51 |
| Figura 22 - Diagrama sobre o funcionamento básico dos DAOs..... | 52 |
| Figura 23 - Método <i>salvar</i> | 53 |
| Figura 24 - Método <i>excluir</i> | 55 |
| Figura 25 - Método <i>buscar</i> | 56 |
| Figura 26 - Método <i>Buscartodos</i> | 56 |
| Figura 27 - Método <i>toValueObject</i> | 57 |
| Figura 28 - Classe <i>DAOModule</i> | 58 |
| Figura 29 - Tela de cadastro de Projeto | 60 |
| Figura 30 - Tela de cadastro de Requisitos do Projeto | 61 |
| Figura 31 - Tela de seleção de <i>stakeholders</i> para o projeto | 62 |
| Figura 33 - Quadro de Scrum na tela principal do sistema | 62 |

SUMÁRIO

| | | |
|----------|---|-----------|
| 1 | INTRODUÇÃO | 11 |
| 1.1 | OBJETIVO GERAL | 11 |
| 1.2 | OBJETIVOS ESPECÍFICOS | 11 |
| 1.3 | JUSTIFICATIVA | 12 |
| 2 | REVISÃO BIBLIOGRÁFICA | 13 |
| 2.1 | SCRUM | 13 |
| 2.2 | NOSQL | 16 |
| 2.3 | MONGODB | 18 |
| 2.3.1 | Sharding | 19 |
| 2.3.2 | Utilização | 20 |
| 2.4 | MORPHIA | 24 |
| 2.5 | INJEÇÃO DE DEPENDÊNCIA | 27 |
| 2.6 | GUICE FRAMEWORK | 27 |
| 2.7 | JAVA | 28 |
| 3 | MATERIAL E MÉTODOS | 29 |
| 3.1 | AMBIENTE DE DESENVOLVIMENTO | 29 |
| 3.2 | ANÁLISE E PROJETO DO SISTEMA | 30 |
| 4 | RESULTADOS E DISCUSSÕES | 39 |
| 4.1 | LIGAÇÃO ENTRE O BANCO DE DADOS E A APLICAÇÃO | 39 |
| 4.1.1 | Value Objects, POJOS e Anotações do Morphia | 40 |
| 4.1.2 | Classe MongoConnection | 44 |
| 4.1.3 | Classes DAO | 48 |
| 4.1.4 | Implementação de o Método Salvar | 52 |
| 4.1.5 | Implementação de o Método Excluir | 55 |
| 4.1.6 | Implementação do método Buscar, BuscarTodos e ToValueObject | 56 |

| | | |
|----------|---|-----------|
| 4.2 | INJEÇÃO DE DEPENDÊNCIA – GUICE | 57 |
| 4.3 | RESULTADOS ALCANÇADOS | 60 |
| 5 | CONSIDERAÇÕES FINAIS | 64 |
| 5.1 | CONCLUSÃO | 64 |
| 5.2 | TRABALHOS FUTUROS/CONTINUAÇÃO DO TRABALHO | 65 |
| | REFERÊNCIAS BIBLIOGRÁFICAS | 66 |

1 INTRODUÇÃO

Segundo Eduardo Bezerra (2007), “O *software* é o combustível utilizado pelos negócios modernos, construir e manter *software* de qualidade, de forma repetível e previsível é difícil hoje e se tornará cada vez mais difícil”. Algumas empresas adotam metodologias e métricas de desenvolvimento que tendem a prever exatamente tudo que possa acontecer. O SCRUM é indicado para situações contrárias a esta, ou seja, situações em que não se tem tempo para prever, ou simplesmente é impossível prever e medir o desenvolvimento do projeto.

Segundo Ken Schwaber (2007), o SCRUM não é uma metodologia, é um *framework*. Isto torna claro que o SCRUM não tem o objetivo de dizer exatamente o que vai acontecer, mas sim facilitar o desenvolvimento do projeto como um todo.

Existem inúmeras ferramentas que servem para o controle do SCRUM, como o *bugzilla*, o *SpiraPlan* e o *Pronto*, porém nem sempre as mesmas têm propósito específico para o SCRUM: muitas vezes englobam várias outras metodologias de desenvolvimento. Tendo em mente a situação proposta, este trabalho apresenta o desenvolvimento de um *software* para o controle de processos, em gestão de projetos, baseados no SCRUM.

1.1 OBJETIVO GERAL

Desenvolver uma aplicação como estudo experimental, para o acompanhamento de um projeto de desenvolvimento de *software* que utilize a metodologia ágil SCRUM, utilizando o banco de dados MongoDB, juntamente com os *frameworks* Morphia e Google Guice.

1.2 OBJETIVOS ESPECÍFICOS

Este trabalho, que trata do desenvolvimento de *software* para a Web, tem como objetivos específicos:

- Desenvolver um referencial teórico sobre SCRUM, linguagem de programação orientada a objetos para *WEB*, *frameworks* e bancos de dados;

- Analisar, projetar e testar um protótipo, como estudo experimental, para o sistema de acompanhamento de SCRUM;

1.3 JUSTIFICATIVA

Tendo em vista que são poucas as ferramentas que auxiliam no controle do SCRUM, e que as poucas que existem são pagas, e às vezes com valores elevados, este trabalho aborda a proposta de criar e apresentar uma ferramenta de controle de SCRUM, totalmente desenvolvida para ambiente WEB, e de fácil entendimento e agilidade nos processos.

Este *software* trabalha totalmente em ambiente web, sendo executado nos mais diversos navegadores, e é de grande ajuda na utilização do SCRUM, pois fornece fácil e rápido acesso ao quadro de SCRUM e todos os requisitos e projetos, facilitando assim todo o trabalho que se tem de controlar o processo de SCRUM.

Pensando em tornar o desenvolvimento mais ágil, foi escolhido o uso do banco de dados *MongoDB*, que utiliza a tecnologia *NoSQL* (*not only SQL*), escolha esta feita devido ao bom desempenho que o banco proporciona.

2 REVISÃO BIBLIOGRÁFICA

Este capítulo apresenta uma revisão bibliográfica de todos os elementos utilizados na redação do trabalho, focando tanto literatura em livros quanto sites e artigos encontrados na *Web*.

2.1 SCRUM

Segundo Ken Schwaber (1995), o *Scrum* é um *framework*¹ para desenvolver e manter produtos complexos. O *Scrum* é formado pelas Equipes de *Scrum* e os papéis, artefatos e regras associadas. Cada componente do *Scrum* serve para um propósito específico e é essencial para o uso do *Scrum*.

O *Scrum* é baseado nas teorias empíricas de controle de processo, que diz que o conhecimento vem da experiência e deve-se tomar decisões baseados no que se conhece. O *Scrum* aplica este conceito de modo iterativo e incremental para aperfeiçoar a previsibilidade e o controle de riscos. Três pilares sustentam a implementação empírica do controle de processo, e são elas a transparência, inspeção e adaptação:

- A transparência diz que aspectos significativos do processo devem ser visíveis para os responsáveis pelos resultados, e a transparência requer que os aspectos sejam definidos em um padrão no qual todos os envolvidos compartilhem um entendimento comum sobre o que esta sendo visto.
- A inspeção diz que os usuários do *Scrum* devem freqüentemente inspecionar os artefatos *Scrum* e o progresso, com objetivo de detectar possíveis variações indesejadas.
- A adaptação diz que se o inspetor determinar que o produto não seja aceitável, o processo e o material que esta sendo processado deve ser ajustado. Este ajuste deve ser feito o quanto antes, para evitar desvios futuros.

¹ *Framework* é um conjunto de classes que colaboram para realizar uma responsabilidade para um domínio de um subsistema da aplicação. Fayad e Schmidt (Magazine Communications of the ACM. 1997)

A Equipe de *Scrum* é formada pelo *Product Owner*, do *Team* e do *Scrum Master*. As equipes são auto organizadas, assim escolhem a melhor forma de realizar o seu trabalho ao invés de serem dirigidas por outros de fora da equipe, e as equipes são Trans-Funcionais². As equipes do *Scrum* produzem iterativamente e incrementalmente, aproveitando ao máximo as oportunidades para realimentação. Entregas incrementais de produtos “Prontos” garantem que sempre uma versão potencialmente útil do produto seja entregue.

O *Product Owner* é responsável por maximizar o valor do produto e do trabalho da equipe do *Scrum*, mas a metodologia utilizada para isto varia amplamente entre organizações, equipes de *Scrum* e indivíduos. No gerenciamento do *backlog* do produto, por exemplo, o *Product Owner* deve:

- Expressar com clareza os itens do *backlog* do produto;
- Ordenar os itens do *backlog* para alcançar melhor os objetivos e missões;
- Garantir o valor do trabalho desempenhado pelo *Team*.
- Garantir que o *backlog* seja visível a todos e mostre no que a equipe trabalhará em seguida;
- Garantir que a equipe de desenvolvimento entenda os itens do *backlog* do produto no nível necessário.

O *Product Owner* é obrigatoriamente uma pessoa, e não um comitê de pessoas. O *Product Owner* pode representar os desejos ou objetivos de um comitê no *backlog* do produto, mas somente o dono do produto é responsável por mudar algo no *backlog*.

O *Team* é responsável por criar uma versão “usável” do produto e potencialmente incremental. Somente os membros da equipe de desenvolvimento podem criar ou incrementar o produto. O tamanho do *Team* deve ser suficientemente pequeno para que se mantenha ágil, e suficientemente grande para poder completar uma parcela de trabalho. Equipes pequenas podem encontrar limitações nas habilidades necessárias para o desenvolvimento e conseqüentemente podem ter dificuldades em entregar um produto potencialmente incremental.

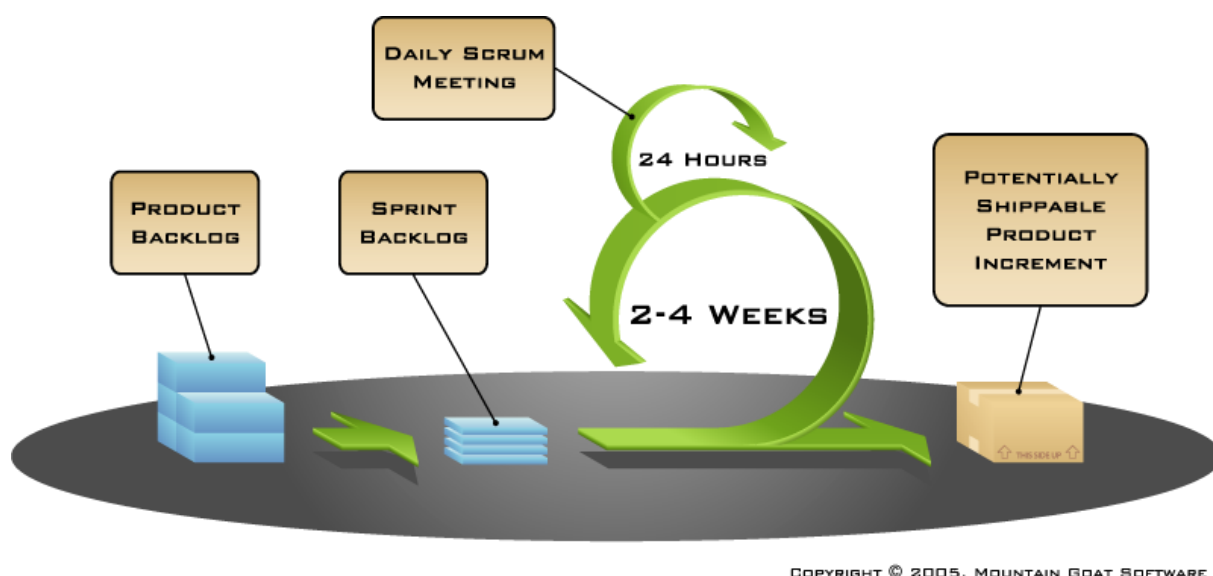
² A própria equipe tem todas as qualificações necessárias para não precisar ser dirigida por nenhuma pessoa externa a equipe

O *Scrum Master* é responsável por garantir que o *Scrum* seja aplicado e entendido corretamente, para isto eles devem garantir que as equipes de *Scrum* obedeçam às teorias, praticas e regras do *Scrum*.

Eventos são usados no *Scrum* para criar uma rotina que minimize a necessidade de encontros não definidos no *Scrum*. Estes eventos de prazo fixo e todos têm prazo de duração máxima, o que garante que um tempo apropriado seja gasto com o projeto evitando perdas de tempo desnecessárias.

Os eventos são uma oportunidade para inspecionar e adaptar algo, e foram feitos justamente para manter e permitir a transparência crítica e inspeção.

Os *Sprints* são eventos com período de um mês ou menos, na qual é criada uma versão potencialmente utilizável do produto. Os *Sprints* têm duração equivalente ao valor de esforço do desenvolvimento, e um *Sprint* começa logo após o termino de outro *Sprint*.



COPYRIGHT © 2005, MOUNTAIN GOAT SOFTWARE

Figura 1 - Ciclo do Scrum
 Fonte: *Mountain goat software* (2007).

Como pode ser visto na Figura 1, o ciclo do SCRUM é relativamente simples, inicialmente começando com a análise do *backlog* do produto já existente (caso exista), para a geração de um novo *Sprint*, definidos na reunião de SCRUM pelo *scrum master* e a equipe do scrum. Durante todo o período do Scrum, que pode durar de duas a quatro semanas, é feito uma reunião diária, para compartilhamento de informações, discussão sobre possíveis problemas encontrados e compartilhar lições aprendidas.

Durante o período de Sprint, os requisitos são controlados de maneira muito ágil, através de *post-its* organizados em um quadro, de acordo com sua prioridade e Status do Requisito. Este quadro é chamado de Quadro de SCRUM.

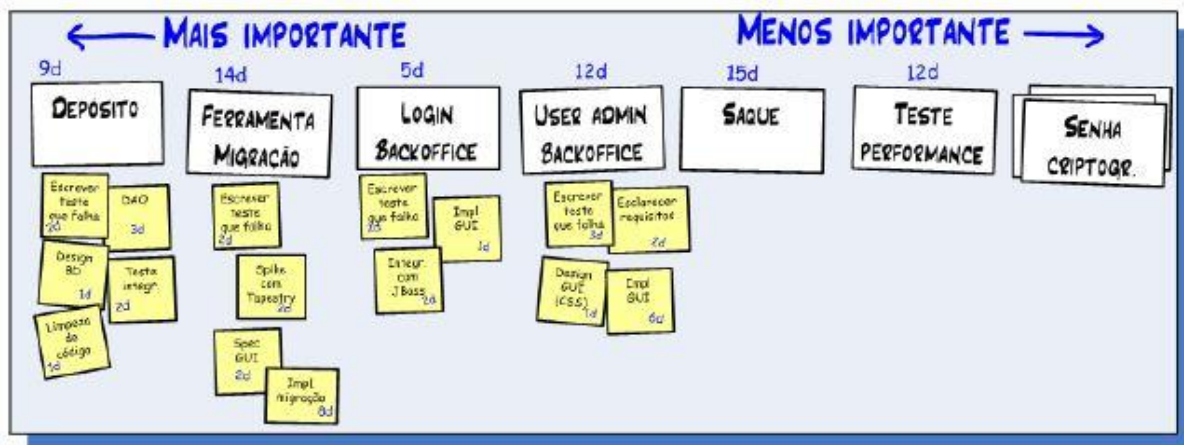


Figura 2 - Exemplo de quadro de Scrum
Fonte: Adaptado de Scrum e XP direto das Trincheiras. (KNIBERG, 2004)

A Figura 2 mostra um exemplo de quadro de Scrum, onde os post-its estão organizados de forma a apresentar os requisitos a serem implementados, da melhor forma possível. A forma de apresentação dos quadros varia de acordo com a necessidade do Team.

2.2 NOSQL

Segundo Alexandre Porcelli (2010), o movimento *NoSQL* (*not only sql*) teve sua origem em 2009, para promover o encontro promovido por Johan Oskarsson e

Eric Evans, que teve como objetivo discutir o crescente surgimento de soluções *Open Source*³ de armazenamento de dados distribuídos não relacionais. Em outubro do mesmo ano, em uma conferência chamada “*no:SQL (east)*”, foi redefinido o uso do termo *NoSQL* para descrever soluções para armazenamento não relacionais.

Pode-se ressaltar que o *NoSQL* não tem o objetivo de invalidar ou substituir totalmente o *SQL*, e sim servir de alternativa e invalidar a idéia de que o modelo relacional é como única solução correta ou válida. (Alexandre Porcelli 2010).

Ao pensar em sistemas de gerenciamento de banco de dados, logo se pensa em tabelas e registros e a necessidade de ter tudo modelado antes da aplicação ser construída. Com *NoSQL*, não necessariamente existem tabelas e registros, o armazenamento é feito de diferentes modos, dependendo do banco de dados *NoSQL* escolhido.

Algumas ferramentas *NoSQL* destacam-se atualmente, são elas:

- Chave-valor: armazena os dados de forma muito semelhante a um *map (java.util.Map)* do Java, assim armazena-se a chave e o valor que pode ser qualquer informação. Todas as buscas são feitas com base na chave, assim todas as chaves devem ser muito bem elaboradas pelo programador;
- Orientado a Documentos: armazena os dados em forma de uma estrutura de dados composta de um número variado de componentes com tipos de dados diversos, da mesma forma que é feito com um arquivo *XML* ou *JSON*. Este modelo tende a incentivar a “desnormalização” dos dados, ou seja, deixando em um só documento todas as informações necessárias para seu funcionamento, e todas as ferramentas orientadas a documento são otimizadas para suportar e aperfeiçoar esta característica;
- Família de colunas (semelhante ao *BigTable*): se tornou popular através do *BigTable* do Google, que foi publicado no ano de 2006. Este modelo é composto por três componentes básicos:
 - *Keyspace* tem como função agrupar Famílias de Colunas, papel bastante semelhante ao de “*Database*” do modelo relacional;

³ *Open Source* é o termo designado para definir qualquer programa que pode ser usado, copiado, estudado e redistribuído sem restrições. Free Software Foundation (<http://www.fsf.org/>, 2011)

- O componente Família de colunas, que tem o papel de armazenar as colunas, muito parecido com as Tabelas do modelo Relacional, mas difere do modelo relacional em vários pontos, como que cada Linha da Família de colunas tem uma chave, aos invés da Família de colunas inteira terem uma chave;
- A coluna é uma tupla formada por nome, *timestamp* e valor onde os dados são realmente armazenados; e
- Grafo: armazenam os dados de forma semelhante à estrutura de dados de uma rede social, onde os dados se ligam aos outros, de forma a criar uma estrutura muito facilmente navegável.

Em consequência da alta escalabilidade, fica praticamente impossível programar as propriedades ACID (Atomicidade, Consistência, Isolamento e Durabilidade) em bancos de dados *NoSQL*. A alternativa as propriedades ACID, foi a implementação de BASE (*Basically, Available, Soft-state, Eventual consistency*). A ideia principal do BASE é abrir mão da consistência em favor da disponibilidade e escalabilidade, ou seja, teoricamente o sistema fica menos seguro e mais escalável e rápido.

Apesar da alta escalabilidade, não é seguro dizer que *NoSQL* pode ser usado em todas as situações, pois muitos sistemas, como por exemplo sistemas bancários, necessitam de 100 por cento de tempo online e não podem ser suscetíveis a falhas, e como a maioria dos bancos de dados *NoSQL* não trabalham com ACID torna inviável sua implementação em certos casos.

2.3 MONGODB

O *MongoDB* é um banco de dados *NoSQL* do tipo orientado a documentos, escrito em C++, e entre suas principais características destacam-se a alta escalabilidade vertical e horizontal, suporte rico em *Queries*, suporte total a índices para todos os campos, *GridFS* que é o armazenamento de arquivos facilitado independente do tamanho. (Mongodb.org 2011)

Segundo Alexandre Porcelli (2010), o *MongoDB* foi criado por Dwight Merriman e Eliot Horowitz, que juntos formaram a 10gen, empresa responsável pelo desenvolvimento e suporte profissional do *MongoDB*.

A estrutura de dados é baseada no modelo *NoSQL* de orientação a documentos, e tem como uma das principais características a capacidade de trabalhar com grandes volumes de dados. Uma particularidade encontrada no *MongoDB* e não muito comum em outros modelos *NoSQL*, são as consultas ricas que podem ser feitas utilizando *MongoDB*.

2.3.1 *Sharding*

Para aplicações grandes e complexas, o *MongoDB* pode ser utilizado sem perder desempenho, devido à tecnologia de *auto-sharding*, que consiste na divisão das tarefas em *shards* horizontais, ou seja, outras máquinas servidoras processando o mesmo banco de dados *MongoDB*. Cada *Shard* é controlado automaticamente pelo *MongoDB* e guarda uma porção do total de dados armazenados, e a leitura e escrita é feita automaticamente no *shard* correspondente aos dados sendo trafegados.

Todos os *Shards* são ligados a uma réplica, que consiste em outro servidor que copia os dados armazenados do *shard*, e atuam como servidores secundários. Caso um *shard* caia, a réplica passa a operar no lugar do *shard* original, até que o mesmo volte a operar. Todas as escritas e leituras consistentes são feitas diretamente nos *shards* primários, e eventualmente todas as leituras consistentes são distribuídas entre as réplicas.

Múltiplos servidores de configuração ficam ativos, os quais guardam as informações sobre qual dado vai para qual *shard*. Os *shards* possuem várias rotas, utilizadas dinamicamente pelo *MongoDB*, e cada rota leva a operações específicas.

2.3.2 Utilização

O primeiro passo para iniciar a utilização do *MongoDB* é a instalação do banco de dados do ambiente, que esta disponível para *download* na pagina oficial do *MongoDB*⁴.

Por padrão o *MongoDB* utiliza o caminho `/data/db`, mas este caminho não é criado automaticamente pelo banco. No diretório raiz onde foi extraído o *MongoDB*, crie a estrutura de pastas `/data/db`. Opcionalmente, pode ser definido o caminho do banco na inicialização do *MongoDB*, passando o parâmetro `--dbpath` na inicialização do `mongod.exe`, mas isto só pode ser feito via *prompt* de comando.

Depois de criada a estrutura de pastas necessária, deve-se executar o arquivo `mongod.exe` para iniciar o serviço do *Mongo*, e a janela vai ficar ativa na tela caso tenha sido iniciado com sucesso. Pode-se também abrir uma interface para administração dos dados pelo `mongo.exe`, no qual se podem inserir bancos e *collections*, assim como inserir documentos e buscar também.

```
F:\Programacao\APIs\mongodb\bin>mongo.exe
MongoDB shell version: 1.8.1
connecting to: test
> use meuBanco
switched to db meuBanco
>
```

Figura 3 - Criação de um banco de dados no *MongoDB*

Para inserir um dado simples com *MongoDB*, deve-se primeiro ter o banco iniciado. Depois entre no `mongo.exe`, que é a interface para uso que vem por padrão no *MongoDB*. Como visto na Figura 3, o comando “use meuBanco”, troca para o db do *test*(que é o padrão), para o “meuBanco”. O Banco “meuBanco” não é automaticamente criado neste momento, e sim no momento que o primeiro dado é inserido no banco.

O *MongoDB* não tem tabelas, como um banco de dados convencional. Os dados são inseridos em documentos, e esses documentos são guardados em *collections*. As *collections* são criadas automaticamente no momento da primeira inserção de documento nela.

⁴ É altamente recomendada a instalação da versão 64 bits, mas o *MongoDB* funciona normalmente tanto na versão 32 bits quanto na versão 64 bits

```
> x = { nome : "pedro" };
< "nome" : "pedro" }
> db.usuarios.save(x);
>
```

Figura 4 - Inserção de dados no banco de dados selecionado

Como pode ser visto na Figura 4, foi criado um documento com nome “x”, na qual foi adicionado um atributo “nome” com o valor “pedro”. Neste caso, x vai ser o documento a ser inserido no banco. Ao executar o comando “`db.usuarios.save(x);`”, como é a primeira inserção no banco “meuBanco”, vai ser criado automaticamente o banco “meuBanco” e a *collection* “usuarios”, e o documento “x” é inserido na *collection* “usuarios”.

Ao executar a inserção, o *prompt* não mostra que foi inserido, então para ter certeza que foi inserido, uma busca deve ser feita, que retornara os dados do documento “x” que foi inserido. Para executar uma busca simples em todos os dados da *collection* “usuarios”, no *prompt* do *MongoDB* digite “`db.usuarios.find();`”

```
> db.usuarios.find();
< "_id" : ObjectId<"4e9c4d3e8609b5fecffa0f20">, "nome" : "pedro" }
> _
```

Figura 5 - Busca com o método *Find*

Como visto na Figura 5, o comando *find* sem receber parâmetros, busca por todos os dados na *collection* da qual foi chamado, no caso a *collection* “usuarios”. Antes na inserção, somente o dado de “nome” havia sido passado, mas agora ao buscar o banco retornou dados de “nome” e “_id”, isto porque o próprio *MongoDB* gera um id automaticamente nas inserções. Caso o id esteja preenchido no documento na hora de salvar, ele salva por cima do documento correspondente ao id, fazendo uma operação de *update*. Deve-se tomar cuidado com inserções com id preenchido, pois o documento anterior com id não pode ser recuperado após uma operação de *save* que passa o id preenchido.

Há casos, em que uma busca de documento com base no id precisará ser feita, para este caso pode-se usar o método “*findOne*” do *MongoDB*, ou simplesmente usando o próprio método *find*, passando para ele um documento com o “_id” preenchido. A diferença nestes dois casos, é que o método *find*, pode retornar mais de um resultado e o método *findOne* retorna obrigatoriamente um

resultado somente. O uso apropriado destes métodos deve ser previamente estudado para melhor aproveitamento do *MongoDB* em cada situação.

Há vários casos também, em que o são necessários buscas mais precisas, passando por parâmetro campos que não é o id, por exemplo, uma busca de todos os usuários que contenham o nome “pedro”. Para estes casos, podem ser feitas *Querys*, que junto com o método *find* filtram os resultados encontrados de acordo com a *Query* que foi feita.

```
> db.usuarios.find({'nome':'pedro'})  
< {"_id" : ObjectId("4e9c4d3e8609b5fecffa0f20"), "nome" : "pedro" }  
>
```

Figura 6 - Busca com Query

Como pode ser visto na Figura 6, foi passado um documento contendo um campo “nome” com valor “pedro” dentro do método *find*, deste modo, a busca retornou todos os documentos que continham nome “pedro”. Foi feito diretamente o objeto dentro do *find*, mas poderia ter sido criado externo e passado como parâmetro, como feito na Figura 6.

```

> x = {"nome": "pedro"};
< {"nome" : "pedro" }
> db.usuarios.find(x);
< {"_id" : ObjectId("4e9c4d3e8609b5fecffa0f20"), "nome" : "pedro" }
>

```

Figura 7 - Query com documento explícito

Utilizando-se de *queries* no *MongoDB*, várias pesquisas ricas podem ser feitas sem utilização de SQL, de maneira muito facilitada e rápida. Podem ser criados documentos complexos e passando como parâmetro para o método *find*. O método mais simples de criar *queries* é passar direto ao método *find*, os campos. Mas pode-se criar *queries* também simplesmente criando um documento e passando ao método *find* como parâmetro.

Podem-se ordenar os resultados usando *sort*, e passando para o *sort* o valor a ser ordenado como parâmetro. Exemplo: “db.usuarios.find().sort({nome:1})”

Usando *\$gt* pode-se pesquisar por todos que sejam Maior que. Exemplo: “db.usuarios.find({idade: {\$gt:33}})” pesquisa por todos na *collection* usuários que tenham idade maior que 33.

Usando *\$ne* pode-se pesquisar por todos excluindo algum resultado da busca. Exemplo: “db.usuarios.find({idade: {\$ne: 33}})” pesquisa por todos na *collection* usuários menos pelos que tenham idade igual a 33.

Usando o caractere “/” antes ou depois da palavra a ser passada como parâmetro a um método *find*, pode pesquisar por partes específicas dentro de uma *query*, semelhante ao comando *like*, que a grande maioria dos bancos de dados relacionais tem. Exemplo: “db.usuarios.find({nome: /Joao/})” faz uma *query* no *MongoDB* equivalente a “select * from usuários where nome = '%Joao%'” em *query* SQL relacional.

Com uso de *Limit* é possível limitar o número de resultados de acordo com sua necessidade. Por exemplo: “db.usuarios.find().limit(10)” pesquisa todos na *collection* usuários mas retorna somente 10 resultados.

Se usado o comando/método *Skip* pode-se definir quantos resultados serão pesquisados além de limitar seus resultados, muito útil para ser utilizado em pesquisas dinâmicas. Por exemplo: “db.usuarios.find().limit(10).skip(20)” vai pesquisar somente até 20 documentos, e retornar os 10 primeiros;

Para saber o número de documentos adicionados a uma *collection*, pode ser usado o método *count*. Por exemplo: “`db.usuarios.count()`” retorna o número de documentos adicionados a *collection* *usuarios*.

Pode-se ainda utilizar o *count* junto com outros métodos na *query*, por exemplo : “`db.usuarios.find({idade: {$gt: 33}}).count()`” retorna o número de usuários que tem idade com maior que 33.

2.4 MORPHIA

O banco de dados *MongoDB* apresenta-se pouco prático quando são necessárias operações mais complexas, pois métodos para inserção, busca e exclusão dos dados devem ser programados, tornando o trabalho com *MongoDB* pouco produtivo. O uso de um *framework* para acesso a dados é indicado, pois facilita o processo, trazendo vários comandos prontos para o acesso a dados. O *framework Morphia* vem como uma alternativa interessante para projetos que envolvem a linguagem de programação Java, pois é leve e tem a finalidade de mapear os Objetos Java para documentos *MongoDB*.

Com o *Morphia* pode-se criar abstrações do banco de dados “*Datastore*” e DAO (*data access object*)⁵ genéricos ou não, para seus objetos. Seu mapeamento no objeto é feito a partir de anotações, tornando a aplicação livre de vários arquivos XML (*Extensible Markup Language*). O *framework Morphia* disponibiliza também classes ricas para criação das *Querys* do *MongoDB* muito facilitadas, tornando a criação de *Querys* muito mais fácil.

Outra grande vantagem é a total integração com outros *frameworks* como *Guice*, *Spring* e outros *frameworks* de Injeção de dependência, como também compatibilidade com GWT (*Google Web Toolkit*)⁶.

Trabalhar utilizando o *framework Morphia* é relativamente simples, e torna o trabalho com *MongoDB* bem mais produtivo, pois ele trabalha com anotações, para fazer o mapeamento das classes *POJO*(*Plain Old Java objects*) da aplicação para

⁵ O padrão de projetos DAO prove uma simples e consistente API para acesso ao banco de dados, sem a necessidade de conhecimento de *frameworks* mais complexos. (<http://www.codefutures.com/java-dao/>, 2011).

⁶ *Framework* que facilita a criação e manutenção de aplicativos *frontend* em *javascript* (<http://code.google.com/intl/pt-BR/webtoolkit/>, 2011).

documentos do *MongoDB*. As seguintes anotações necessárias nas classes *POJOs* trabalharem com o *Morphia*:

- *@Entity* – Anotação que deve ser feita na classe *POJO*. Entre seus atributos é importante ressaltar o atributo *name*, que é o nome na qual vai ser nomeada a *collection* no *MongoDB*, caso o atributo não seja declarado o nome da *collection* vai ser o nome da classe. Caso o *POJO* não seja anotado com *@Entity* a classe não vai ser mapeada pelo *Morphia* e conseqüentemente não vai ser usada pelo *MongoDB*.
- *@Indexes* – Anotação usada na classe, e consiste em uma lista dos índices da classe, caso não queira que os índices sejam feitos nos campos. Esta anotação é opcional, pois não é obrigatório ter índices nos campos (apesar de ser recomendado).

Com estas anotações somente já é possível salvar uma classes *POJO* simples, depois de configurados os *DAOS* e a conexão com o banco. Para um controle mais preciso é necessário o uso de anotações nos campos da classe:

- *@Transient* – Indica que o campo não será salvo no banco de dados.
- *@Serialized* – Indica que o campo será convertido para dados binários ao ser persistido.
- *@NotSaved* – Indica que o campo não será salvo, mas poderá ser buscado pelo *Morphia*.
- *@AlsoLoad* – Indica que o campo pode ser buscado com qualquer nome passado por parâmetro. Basicamente burla o “*caseSensitive*”, bom para migrações de bases.
- *@Indexed* – Indica que o campo vai ser indexado pelo *MongoDB*, assim a busca por esse campo fica mais rápida devido à arquitetura do *MongoDB*.
- *@Version* – Campo de controle de versão otimista. É controlado automaticamente pelo *Morphia*, não necessitando nenhuma implementação extra.
- *@Reference* – Indica que o campo é uma referencia a outro documento. Alguns valores podem ser definidos a esta anotação, entre elas alguns dos mais importantes são o *Lazy*, que indica que o campo referenciado vai ser carregado na próxima chamada de método da

instancia do *Proxy*, o *ignoreMissing* indica que não será gerada *exception* caso não for possível carregar a referencia, e o *concreteClass* indica qual é o *class TYPE* da classe referenciada.

- *@Embedded* – Por padrão os campos são todos *Embedded*, mas anota-los explicitamente possibilita algumas customizações no campo, como por exemplo mudar o nome do atributo que vai ser salvo na *collection*.

Existem também anotações que controlam o ciclo de vida dos objetos, anotações estas que com o uso certo podem ser muito uteis para o sistema. Estes métodos obrigatoriamente devem ser implementados dentro da classe com a anotação *@Entity*:

- *@PrePersist* - O método anotado é chamado antes de o *POJO* ser salvo.
- *@PostPersist* – O método anotado é chamado após o *POJO* ter sido salvo.
- *@PreLoad* - O método anotado é chamado antes de o objeto ser carregado no *POJO*, pode receber parâmetros para adicionar/remover/editar o objeto.
- *@PostLoad* – O método anotado é chamado após o objeto ser carregado no *POJO*.

Estes métodos não podem executar funções de deletar, pois quebra a segurança interna do próprio *framework Morphia*. Por padrão o *Morphia* tenta converter todos os tipos primitivos dos objetos para tipos suportados pelo *MongoDB*. Os tipos suportados pelo *MongoDB* são quatro:

- *Integer* (32 bit);
- *Long* (64 bit);
- *Double*(64 bit);
- *String*.

Alguns outros tipos também são convertidos com sucesso para tipos do *MongoDB*, são eles :

- *enum* – todo os *enums* são convertidos para *String* no *MongoDB*;
- *java.util.Date* - é convertido para ms (milisegundo) desde época UTC(Tempo universal Coordenado);

- *java.util.Locale* – é convertido para String;
- *com.mongodb.DBRef* - é armazenado sem conversão, pois tem compatibilidade nativa com o *MongoDB*;
- *com.mongodb.ObjectId* – é armazenado sem conversão, pois tem compatibilidade nativa com o *MongoDB*.

Outros tipos os quais não serão convertidos devem ser devidamente adaptados antes de serem salvos para evitar *exceptions*.

2.5 INJEÇÃO DE DEPENDÊNCIA

Segundo Magno Machado (Magno Machado, 2011), "a inversão de controle ocorre sempre que o fluxo de ação sai do nosso controle". Analisando, pode-se dizer que a inversão de controle ocorre sempre que é deixado para um *framework* ser responsável pela execução de uma ou várias partes do código. Então, pode-se dizer que não existe um *framework* para Inversão de Controle, pois é muito comum deixarmos certas partes do código ser controladas por *frameworks* ou às vezes até biblioteca de classes.

Segundo Martin Fowler (Martin Fowler, 2011), "não se pode dizer que um *framework* é especial porque tem inversão de controle, pois seria como dizer que um carro é especial porque possui rodas." Assim, pode-se definir que injeção de dependência é um tipo de inversão de controle, pois com injeção de dependência, injeta-se automaticamente nas classes mapeadas as suas dependências, tirando esta responsabilidade do programado e passando a responsabilidade ao *framework*, ocorrendo neste momento uma inversão de controle.

2.6 GUICE FRAMEWORK

Segundo VANBRABANT (2008), "usar injeção de dependência freqüentemente significa que, ao invés de colocar suas dependências, você opta por receber estas dependências de algum lugar, e você não se importa com a origem delas", e esta é a idéia básica da injeção de dependência que o Google Guice proporciona.

Como visto no site oficial do Google Guice (code.google.com/p/google-guice, 2011), o Guice *Framework* simplesmente alivia o uso de *Factories* e o uso de *new* no código Java. Pode-se dizer que o *@Inject* do Guice é o novo *new* (*Guice's @Inject is the new new*). Ainda tem-se a necessidade de usar *Factories* em alguns casos, mas seu código não depende mais diretamente deles, tornando o código mais fácil de ser modificada, melhora a reusabilidade e os testes unitários.

O Google utiliza-se do Google Guice em sistemas missão Crítica (casos em que o sistema deve ficar online 24 horas por dia, 7 dias por semana) desde o ano de 2006.

Com a Injeção de Dependência, os objetos aceitam as dependências nos seus construtores. Para construir um objeto, primeiro constroem-se suas dependências, porém as dependências também tem suas dependências, assim acaba-se criando um grafo de objetos. Construir grafos de objetos manualmente é uma tarefa muito demorada, suscetível a erros e difícil de testar, porém, o Google Guice constrói automaticamente os grafos de objetos.

2.7 JAVA

Segundo Laura Lemay (1999), “Java é uma linguagem de programação orientada a objetos desenvolvida pela Sun Microsystems”. Foi modelada baseada no C++, e foi feita com intuito de ser simples, pequena, e portátil através das plataformas e sistemas operacionais. Ser independente de plataforma é umas das significativas vantagens do Java sobre outras linguagens de programação, especialmente para sistemas que necessitam trabalhar em muitas plataformas diferentes.

Os tipos primitivos do Java têm tamanhos consistentes nas mais diferentes plataformas, e as bibliotecas de classes do Java facilitam muito a programação das classes, eliminando a necessidade de reescrever todo o código quando migrada ou adaptada para outra plataforma. A independência de plataforma não é somente no nível de código, pois os arquivos binários do Java são transformados em são transformados em *bytecodes*, na qual a *JVM(Java virtual machine)* interpreta os dados e compila de forma a que possa funcionar nas mais diversas plataformas.

3 MATERIAL E MÉTODOS

O trabalho desenvolvido é composto primeiramente por uma revisão bibliográfica dos recursos utilizados, sendo esses: SCRUM, MongoDB, Morphia, Google Guice e Java. O sucesso nesta etapa é importante, pois segundo Silva e Menezes (2001), “a revisão de literatura é fundamental, porque fornecerá elementos para evitar a duplicação de pesquisas sobre o mesmo enfoque do tema, e favorecerá a definição de contornos mais precisos do problema a ser estudado”.

O sistema proposto neste trabalho foi desenvolvido em conjunto com Carlos Alexandro Becker, o qual implementou toda a parte Cliente da aplicação, programando as telas e fazendo a ligação com a parte Servidor da aplicação, tema deste trabalho.

3.1 AMBIENTE DE DESENVOLVIMENTO

O *eclipse* foi utilizado como editor de código JAVA e compilador. O eclipse é uma comunidade *Opensource* focada em desenvolver uma plataforma de desenvolvimento composta por vários *frameworks*, ferramentas e *runtimes* para construção, *deploy* e gerenciamento de software. Juntamente com o eclipse foi utilizado o *plugin* do GWT, e os JARs dos *frameworks* necessários.

Para a modelagem UML (*Unified Modeling Language*), foi utilizada a ferramenta *Visual Paradigm*. O *Visual Paradigm* fornece um conjunto de produtos premiados que facilita as organizações a projetar visualmente e esquematicamente, integrar e implementar suas aplicações corporativas de missão crítica e suas bases de dados subjacentes (VISUAL PARADIGM, 200-).

Para prover um controle de alterações no projeto, foi utilizado o Google code como repositório SVN, e por o *plugin* para o eclipse Subversive.

3.2 ANÁLISE E PROJETO DO SISTEMA

O sistema de controle de *Scrum*, que foi nomeado de GPA (acrônimo para Gerenciador de Projetos Ágeis), foi desenvolvido em conjunto com o Carlos Alexandro Becker, e o sistema foi desenvolvido utilizando-se de várias tecnologias atuais, como *GWT*, *MongoDB*, Google Guice, *GWTPlatform* entre outras.

Neste trabalho, só serão abordados assuntos relacionados à parte “servidor” da aplicação, ou seja, como foi estruturado o banco de dados do sistema, e a ligação entre a parte “cliente” do sistema, com a parte “servidor”.

Como o *GWT* na parte cliente tem suas limitações e nem todas as classes podem ser utilizadas, algumas adaptações e padrões de projetos foram utilizados para fazer o sistema trabalhar de modo que a parte cliente e servidor trabalhem sem conflitos.

O sistema tem como objetivo principal, o cadastro e controle dos projetos, requisitos e *stakeholders* envolvidos, assim como o controle de fluxo dos encaminhamentos de requisitos por meio de um quadro virtual, para representar um quadro de *Scrum* real, onde o projeto é selecionado, e os requisitos dele são apresentados em forma de “*post-it*”.

O quadro de *Scrum* contém quatro estados chamados de Status do Requisito e são eles: aguardando, desenvolvimento, testes e concluído. Um destes quatro Status sempre estará salvo em cada encaminhamento de um requisito. Quando o requisito é criado, será criado um encaminhamento padrão para aguardando, assim o requisito sempre estará no quadro de *Scrum* do projeto correspondente.

A **Erro! Fonte de referência não encontrada.** mostra o MER (Modelo Entidade Relacionamento) do sistema, com suas entidades e ligações. Os cadastros básicos do sistema, no primeiro momento, serão quatro: usuário, *stakeholder*, produto e projeto.

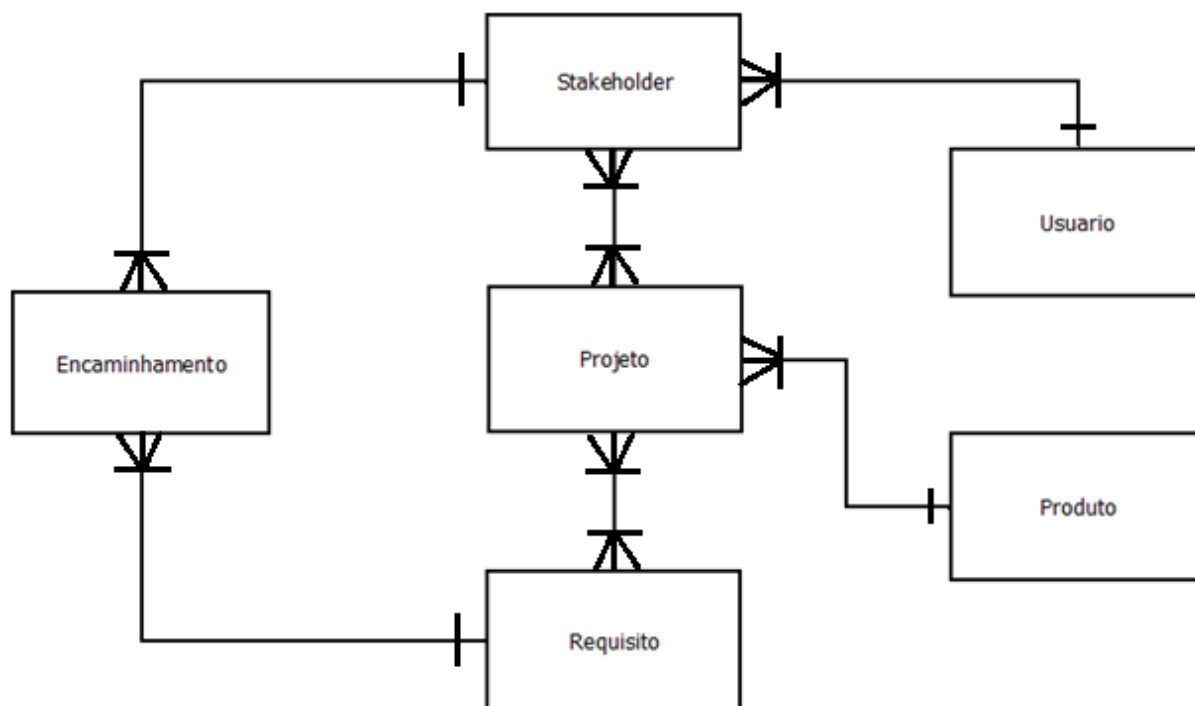


Figura 8 - MER da aplicação

O cadastro do usuário tem objetivo de salvar os dados para *login*, nome e informações relevantes a usuários do sistema. A *collection* que guardará os dados tem nome de “usuários”, e seus campos serão os seguintes:

- *Id* – campo que guardará o ID da *collection* *usuarios*. Seu tipo será *ObjectID*, que é uma classe da API do *driver* Java do *MongoDB* que tem objetivo de guardar as informações de id, e gerar automaticamente este valor para novos registros.
- *Login* – campo que combinado com uma senha, dará acesso para o usuário ao sistema. Seu tipo é *String* e seu valor é único em todo sistema, ou seja, não terão cadastrados dois usuários com o mesmo *login*.
- *Senha* – campo que combinado com um *login*, dará acesso para o usuário ao sistema. Seu tipo é *String*.
- *Nome* – campo de tipo *String*, contendo informações de nome do usuário.

- Administrador – campo *boolean* que tem objetivo de dizer se o usuário tem permissões de administrador. Somente usuários administradores poderão cadastrar novos projetos, e pesquisar por todos os projetos.

O cadastro de *stakeholder* tem por objetivo cadastrar os usuários, que serão diretamente envolvidos em cada projeto. Os campos envolvidos na *collection stakeholders* são:

- Id – Campo de tipo *ObjectID*, que é controlado pelo *MongoDB*, e pode ser usado para buscas, pois é o identificador único da *collection stakeholders*.
- Nome – Campo do tipo *String* que guardará dados do nome do stakeholder.
- Papel *Stakeholder* – Campo que indica qual será o papel realizado pelo *stakeholder* no projeto. Os papéis de *stakeholder* que serão disponíveis no sistema serão: dono, gerente de projeto, fornecedor de requisitos, desenvolvedor, *tester*, analista de sistemas e analista de negócios.
- Usuário – é ligado a *collection* “*usuarios*”. Guarda informação de qual usuário é responsável por este *stakeholder*.

O cadastro de produto tem intuito simples de guardar informações sobre um produto, o qual será referenciado no cadastro de projetos. Um produto pode ter vários projetos sendo executados sobre ele, e obrigatoriamente todo projeto deve ser executado para um produto. Inicialmente, o cadastro de produtos terá somente dois campos, o Id e o Nome.

O cadastro de projeto é à base de todo sistema. A partir do cadastro de projeto, produtos serão relacionados, os *stakeholders* existentes serão selecionados, e o quadro de scrum será gerado. Os campos do cadastro de projeto são os seguintes:

- Id – identificador único da *collection* “*projetos*”. Seu tipo é *ObjectID* tal como os Ids das outras *collections* do sistema.
- Nome – campos *String*, único em todo sistema, ou seja, não poderão ser cadastrados dois projetos com o mesmo nome.
- Data Início – campo do tipo *Date*, que guarda a data em que o projeto terá início.

- Data Fim – campo do tipo *Date*, que guarda a data em que o projeto será finalizado.
- Stakeholders – campo do tipo *List*, que guarda todos os stakeholders envolvidos no projeto.
- Produto – campo ligado a *collection* “produtos” que indica qual produto esta sendo criado/modificado pelo projeto.

A partir do cadastro de projeto um novo cadastro será habilitado, o cadastro de requisito. Os requisitos serão os passos que os *stakeholders* terão que tomar para a realização do projeto, e cada requisito estará ligado a um projeto, e também guardará os dados de encaminhamentos a serem. Os dados da *collection* “requisitos” serão:

- Id – identificador único do cadastro de requisito. Seu tipo é *ObjectID* e é o identificador único da *collection* “requisitos”.
- Titulo – campo *String* que guarda o titulo do projeto. O título do projeto é usado para identificar o projeto na lista de projetos do usuário, na tela principal, e é usado na busca.
- Descrição – campo *String* que guarda informações sobre o projeto.
- Prioridade Requisito – campo que guarda informações referentes à prioridade em que o requisito deve ser implementado. A partir deste dado, a cor do “*post-it*” muda. As prioridades requisito presentes no sistema serão Alto, Médio e Baixo, posteriormente podendo haver outros requisitos implementados. Seu tipo é um *Enum*.
- Tempo Estimado – campo do tipo *Integer* que guarda o tempo estimado para desenvolvimento do requisito, em horas.
- Encaminhamentos – campo do tipo *List*, que guarda os encaminhamentos do requisito. No momento em que o requisito é cadastrado no sistema, um requisito padrão é cadastrado.
- Data Cadastro – campo do tipo *Date*, que guarda a data em que o requisito foi cadastrado no sistema. É preenchido automaticamente pelo sistema.
- Tempo Total – campo do tipo *Integer*, que indica o tempo total utilizado para realização do requisito, em horas.

- Projeto – campo ligado a `collection` “projetos” que indica o projeto que terá este requisito implementado.

A partir do requisito, os encaminhamentos serão gerados. Encaminhamentos indicarão em que estado anda o desenvolvimento de cada requisito, assim possibilitando o controle básico do quadro de scrum.

Todo encaminhamento terá um campo chamado `StatusRequisito`, que indica em qual estado o requisito se encontra atualmente. O campo `StatusRequisito` é um *enum* e os estados disponíveis no sistema são:

- Aguardando – indica que o requisito ainda não foi escolhido por nenhum *stakeholder* para nenhuma tarefa.
- Desenvolvendo – indica que o requisito se encontra em desenvolvimento.
- Testando – indica que o desenvolvimento do requisito já foi concluído, e testes estão sendo feitos sobre aquele requisito.
- Concluído – indica que o requisito já foi desenvolvido e testado, e passou nos testes.

A *collection* de encaminhamentos conta ainda com outros campos usados para controle e melhor detalhamento do estado em que o requisito se encontra. São eles:

- Id – índice único da *collection* “Encaminhamentos”.
- *Stakeholder* - ligado a *collection* “stakeholders”, guarda a informação sobre qual é o stakeholder responsável pelo encaminhamento, no presente `StatusRequisito`.
- Data – campo do tipo *Date* que indica qual a data em que o encaminhamento foi criado.
- Descrição – campo do tipo *String* que armazena informações sobre o encaminhamento.

No sistema, cada *stakeholder* estará ligado a um usuário, mas o usuário não vai estar ligado a nenhum stakeholder, de modo que um único usuário possa estar em mais de um stakeholder. Cada projeto vai ter um produto, mas o produto não vai ter adicionado internamente o projeto, cada projeto tem também um ou mais stakeholders e um ou mais requisitos adicionados. Cada requisito terá um ou mais

encaminhamentos, e cada encaminhamento terá obrigatoriamente ligação com um *stakeholder*.

As Classes geradas na implementação do sistema foram muitas, tornando muito difícil a geração de um diagrama de classes do sistema, por este motivo será somente citadas as classes de acordo com seus pacote correspondente. As classes internas não serão citadas.

No pacote `com.geekvigarista.scrummanager.server.actionhandlers` foram implementadas as classes necessárias para a ligação com a parte cliente da aplicação:

- `cadastros.encaminhamento.ExcluirEncaminhamentoActionHandler`
- `cadastros.encaminhamento.SalvarEncaminhamentoActionHandler`
- `cadastros.produto.BuscarTodosProdutosActionHandler`
- `cadastros.produto.LoadProdutoActionHandler`
- `cadastros.produto.SalvarProdutoActionHandler`
- `cadastros.projeto.BuscarProjetoLikeActionHandler`
- `cadastros.projeto.BuscarProjetosByUsuarioActionHandler`
- `cadastros.projeto.CarregarRequisitosNoProjetoActionHandler`
- `cadastros.projeto.LoadProjetoActionHandler`
- `cadastros.projeto.SalvarProjetoActionHandler`
- `cadastros.requisito.ExcluirRequisitoActionHandler`
- `cadastros.requisito.LoadRequisitoActionHandler`
- `cadastros.requisito.SalvarRequisitoActionHandler`
- `cadastros.stakeholder.BuscarStakeholderActionHandler`
- `cadastros.stakeholder.ExcluirStakeholderActionHandler`
- `cadastros.stakeholder.LoadStakeholderActionHandler`
- `cadastros.stakeholder.SalvarStakeholderActionHandler`
- `cadastros.usuario.BuscarUsuarioActionHandler`
- `cadastros.usuario.ExcluirUsuarioActionHandler`
- `cadastros.usuario.LoadUsuarioActionHandler`
- `cadastros.usuario.login.LoginHandler`
- `cadastros.usuario.login.LogoutHandler`
- `cadastros.usuario.login.VerificaUsuarioLogadoHandler`
- `cadastros.usuario.SalvarUsuarioActionHandler`

No pacote `com.geekvigarista.scrummanager.server.beans` foram implementadas os *beans* com anotações do *Morphia*, usados na persistência:

- `EncaminhamentoPOJO`
- `ProdutoPOJO`

- ProjetoPOJO
- RequisitoPOJO
- StakeholderPOJO
- UsuarioPOJO

No pacote `com.geekvigarista.scrummanager.server.filters` somente a classe `CacheFilter` foi necessária.

No pacote `com.geekvigarista.scrummanager.server.guice` foram implementadas as classes responsáveis pela configuração da injeção de dependência na parte servidor:

- `DAOModule`
- `DispatchServletModule`
- `GuiceServletConfig`
- `ServerModule`

No pacote `com.geekvigarista.scrummanager.server.interfaces` foram armazenadas as interfaces usadas pelo DAO:

- `dao.IDao`
- `dao.IDaoEncaminhamento`
- `dao.IDaoProduto`
- `dao.IDaoProjeto`
- `dao.IDaoRequisito`
- `dao.IDaoStakeholder`
- `dao.IDaoUsuario`

No pacote `com.geekvigarista.scrummanager.server.persistencia` as classes responsáveis efetivamente pela persistência foram salvas:

- `dao.DaoEncaminhamento`
- `dao.DaoProduto`
- `dao.DaoProjeto`
- `dao.DaoRequisito`
- `dao.DaoStakeholder`
- `dao.DaoUsuario`
- `utils.MongoConnection`

Já no pacote `com.geekvigarista.scrummanager.shared` foram armazenadas todas as classes usadas tanto na parte servidor quanto na parte cliente da aplicação:

- `commands.encaminhamento.excluir.ExcluirEncaminhamentoAction`
- `commands.encaminhamento.excluir.ExcluirEncaminhamentoResult`
- `commands.encaminhamento.salvar.SalvarEncaminhamentoAction`
- `commands.encaminhamento.salvar.SalvarEncaminhamentoResult`
- `commands.produto.busca.BuscarProdutoListResult`
- `commands.produto.busca.BuscaTodosProdutosAction`
- `commands.produto.load.LoadProdutoAction`
- `commands.produto.load.LoadProdutoResult`
- `commands.produto.salvar.SalvarProdutoAction`
- `commands.produto.salvar.SalvarProdutoResult`
- `commands.projeto.load.BuscarProjetoLikeAction`
- `commands.projeto.load.BuscarProjetoListResult`
- `commands.projeto.load.BuscarProjetosByUsuarioAction`
- `commands.projeto.load.CarregarRequisitosNoProjetoAction`
- `commands.projeto.load.LoadProjetoAction`
- `commands.projeto.load.LoadProjetoResult`
- `commands.projeto.salvar.SalvarProjetoAction`
- `commands.projeto.salvar.SalvarProjetoResult`
- `commands.requisito.buscar.BuscarRequisitoByIdAction`
- `commands.requisito.buscar.BuscarRequisitoObjResult`
- `commands.requisito.excluir.ExcluirRequisitoAction`
- `commands.requisito.excluir.ExcluirRequisitoResult`
- `commands.requisito.salvar.SalvarRequisitoAction`
- `commands.requisito.salvar.SalvarRequisitoResult`
- `commands.stakeholder.buscar.BuscarStakeholderAction`
- `commands.stakeholder.buscar.BuscarStakeholderByIdAction`
- `commands.stakeholder.buscar.BuscarStakeholderListResult`
- `commands.stakeholder.buscar.BuscarStakeholderObjResult`
- `commands.stakeholder.excluir.ExcluirStakeholderAction`
- `commands.stakeholder.excluir.ExcluirStakeholderResult`
- `commands.stakeholder.salvar.SalvarStakeholderAction`
- `commands.stakeholder.salvar.SalvarStakeholderResult`
- `commands.usuario.buscar.BuscarUsuarioAction`
- `commands.usuario.buscar.BuscarUsuarioByIdAction`
- `commands.usuario.buscar.BuscarUsuarioListResult`
- `commands.usuario.buscar.BuscarUsuarioObjResult`
- `commands.usuario.excluir.ExcluirUsuarioAction`
- `commands.usuario.excluir.ExcluirUsuarioResult`
- `commands.usuario.login.LoginUsuarioAction`
- `commands.usuario.login.LogoutUsuarioAction`
- `commands.usuario.login.VerificaUsuarioLogadoAction`
- `commands.usuario.salvar.SalvarUsuarioAction`

- `commands.usuario.salvar.SalvarUsuarioResult`
- `dtos.ProjetoStakeholderDTO`
- `enums.AcaoEncaminhar`
- `enums.PapelStakeholder`
- `enums.PrioridadeRequisito`
- `enums.StatusRequisito`
- `utils.EncaminharUtil`
- `vos.Encaminhamento`
- `vos.Produto`
- `vos.Projeto`
- `vos.Requisito`
- `vos.Stakeholder`
- `vos.Usuario`

4 RESULTADOS E DISCUSSÕES

Este capítulo apresenta como é feita a ligação entre o banco de dados MongoDB e uma aplicação, pois a demonstração de utilização deste banco de dados é o intuito do trabalho. Para isso, foi utilizado o estudo experimental (que é desenvolvido em conjunto) e apresentada toda a codificação necessária. Em cada item são discutidos os resultados obtidos, a partir do processo de criação e utilização dos elementos necessários para o desenvolvimento e ao final é apresentado o resultado final do sistema, com imagens das telas do sistema, e explicação do funcionamento das principais telas do sistema.

4.1 LIGAÇÃO ENTRE O BANCO DE DADOS E A APLICAÇÃO

O banco de dados utilizado na aplicação não é do modelo relacional, ou seja, não tem tabelas, registros e nem *schema*. O banco é o *MongoDB*, que é uma das ferramentas *NoSQL* existentes, e foi escolhido devida a facilidade em entender a orientação a documentos utilizada pelo *MongoDB*.

O banco por si só trabalha automaticamente com o sistema, assim foi utilizado um *driver* para comunicação do *MongoDB* com o Java. Este *driver* é disponibilizado no próprio site do *MongoDB* em forma de *JAR*, e com este *JAR* na aplicação, o programador tem acesso a classes que facilitam em muito a programação com utilização de *MongoDB*.

Porém, para facilitar ainda mais a produção e ganhar tempo, foi utilizado um *framework* para mapeamento dos objetos da aplicação para o *MongoDB*. O *framework* escolhido foi o *Morphia*, que é disponibilizado em forma de um *JAR* pequeno e leve para a aplicação.

Na parte cliente da aplicação foi utilizado à linguagem *GWT*, e para comunicação da parte Cliente com a parte Servidor, foi utilizado o *framework* de injeção de dependência Google Guice. Este *framework* trabalha com a finalidade de injetar as dependências da parte cliente com a parte servidor, assim ligando minha aplicação sem necessidade de codificação manual suscetível a erros.

4.1.1 Value Objects, POJOS e Anotações do Morphia

Inicialmente, o primeiro passo para usar *Morphia* com *MongoDB* é mapear os *Beans* do sistema com as anotações correspondentes do *Morphia*. Porém, o *Morphia* obriga que cada *Bean* tenha um campo *id* de tipo *ObjectID*, e este *Id* não é compatível com a parte cliente do *GWT* gerando *exceptions* que impossibilitavam o trabalho conjunto do *GWT* com o *Morphia*.

Pensando em resolver este problema, foi utilizado o padrão de projeto *Value Object*, que seria um container puro do *Bean*. Todos os *Value Objects* do sistema, contem somente os campos normais do *Bean*, sem anotações, e seu *id* é do tipo *String*, pois a conversão de *ObjectID* para *String* é muito boa, e *String* é um tipo compatível com a parte cliente do *GWT*.

Todos os *Value Object* implementados implementam a interface *Serializable*, pois eles terão contato direto com o *GWT*, e sem *Serializable* o *GWT* não trabalha corretamente. O *Value Object* de *Produto*, utilizado no sistema, pode ser visto na Figura 9.

```
public class Produto implements Serializable
{
    private static final long serialVersionUID = 1L;

    private String id;
    private String descricao;

    public Produto() {}

    public Produto(String descricao)
    {
        super();
        this.descricao = descricao;
    }

    public Produto(String id, String descricao)
    {
        super();
        this.id = id;
        this.descricao = descricao;
    }

    // Gets e sets ...
}
```

Figura 9 - *Value Object* de produto

Mas somente com os *Value Objects* no sistema, a parte cliente funcionaria completamente, mas o *Morphia* não consegue se conectar com o *MongoDB*, devido à falta de suas anotações e de um id de tipo *ObjectId*. Por este motivo, foi criado na parte servidor da aplicação, vários *POJOS*, os quais podem ser anotados com as anotações do *Morphia*, e serão utilizados exclusivamente nas classes de persistência do sistema (as classes *DAO*).

```
@Entity("produtos")
public class ProdutoPOJO
{
    @Id
    ObjectId id;
    private String descricao;

    public ProdutoPOJO()
    {
    }

    public ProdutoPOJO(Produto produto)
    {
        if(produto == null)
        {
            this.id = null;
            this.descricao = null;
        }
        else
        {
            if(produto.getId() != null)
            {
                this.id = new ObjectId(produto.getId());
            }
            this.descricao = produto.getDescricao();
        }
    }

    // ...
}
```

Figura 10 - Produto Pojo

Como pode ser visto na Figura 10, a classe `ProdutoPojo` apresentada, tem a anotação `@Entity` recebendo como valor a *String* “produtos”, isto indica que todos os objetos `ProdutoPojo` serão salvos em uma coleção de nome “produtos”. Logo abaixo, percebe-se a anotação `@Id` em um objeto de tipo *ObjectId*, necessário pelo *Morphia* para gerar automaticamente os ids do *MongoDB*. O construtor vazio é necessário, pois o *Morphia* o usa internamente em seus métodos, sem ele a classe gera uma *exception* durante a execução.

O outro construtor de `ProdutoPOJO`, que recebe um `Produto` (*Value Object*) como parâmetro, é a chave de todo o funcionamento da arquitetura do banco de dados. Pois, ele pega os valores do *Value Object*, e insere os valores nos campos do próprio `ProdutoPOJO`, basicamente convertendo `Produto` (*Value Object*) para `ProdutoPOJO` (*POJO*).

Em outros *POJOS* do sistema, fora criado um objeto global privado contendo a própria classe, para facilitar a conversão, devido ao número maior de campos e maior complexidade, como pode ser visto na Figura 11.

```

@Transient
private List<RequisitoPOJO> requisitos;

@Transient
private Projeto projeto;

public ProjetoPOJO()
{
}

/**
 * @param projeto
 */
public ProjetoPOJO(Projeto projeto)
{
    super();
    if(projeto == null)
        projeto = new Projeto();
    this.projeto = projeto;
    this.dataFim = projeto.getDataFim();
    this.dataInicio = projeto.getDataInicio();
    this.nome = projeto.getNome();
    this.produto = new ProdutoPOJO(projeto.getProduto());
    if(projeto.getRequisitos() != null)
    {
        this.requisitos = new ArrayList<RequisitoPOJO>();
        for(Requisito r : projeto.getRequisitos())
        {
            RequisitoPOJO rPojo = new RequisitoPOJO(r);
            this.requisitos.add(rPojo);
        }
    }
}

```

Figura 11 - Objeto Transiente para conversão de *POJO* para *Value Object*

Outra funcionalidade de extrema importância nas classes *POJO* é o método que permite retornar um *Value Object* a partir do próprio *POJO*, este método foi chamado `getObjeto` (objeto este sendo substituído pelo nome do *Value Object*, por

exemplo, no ProdutoPOJO, o método é chamado getProduto, já no ProjetoPOJO o método é chamado getProjeto, e assim sucessivamente).

```

public Produto getProduto()
{
    if(this.id != null)
    {
        return new Produto(this.id.toString(), this.descricao);
    }
    else
    {
        return new Produto(this.descricao);
    }
}

```

Figura 12 - método de conversão de POJO para Value Object

Como pode ser visto na Figura 12, o método *getObjeto*, da classe ProdutoPOJO é o *getProduto*, e este método faz a conversão de ProdutoPOJO para Produto(Value Object) chamando o construtor do próprio Value Object usando os campos preenchidos do ProdutoPOJO como parâmetros.

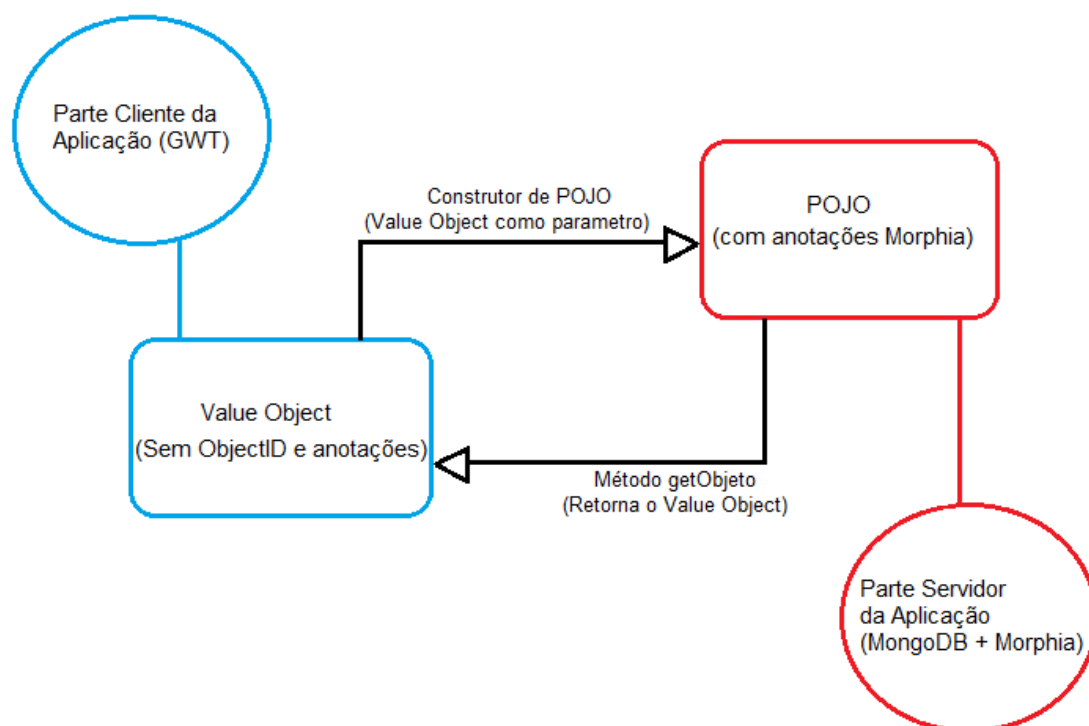


Figura 13- Diagrama de funcionamento da conversão de POJO e Value Object

A Figura 13, explica de modo simples o funcionamento total da conversão de Value Object para POJO, para funcionamento do sistema. Na figura, a parte azul é a

parte Cliente, que não aceita *ObjectID*, e a parte vermelha é a parte Servidor, na qual tem anotações e *ObjectID*.

Ao sair da parte cliente, o objeto está na forma de *Value Object*, então é passado ao servidor e ao ser usado, é convertido para *POJO*, pelo construtor da classe *POJO* correspondente.

Depois de convertido para *POJO*, o objeto pode ser usado para operações no banco de dados, como salvar, excluir e buscar documentos do banco de dados. (lembrando que é tratado como “documento” e não como “registro” do banco de dados, porque *MongoDB* trabalha com orientação a documentos e a palavra “registro” seria menos indicada nesta situação).

Após a operação feita, caso o retorno será um objeto que terá de ser usado na parte cliente, após uma operação de busca, por exemplo, o objeto vem do banco na forma de *POJO*, então uma conversão para *ValueObject* é necessária, conversão esta que é feita no método *getObjeto*.

4.1.2 Classe *MongoConnection*

Tendo os *Value Objects* criados, e os *POJOs* devidamente anotados e implementados, o próximo passo para utilização de *MongoDB* e *Morphia* no sistema é a criação de uma classe utilitária para ligação direta com o banco de dados. Esta classe foi criada com nome de “*MongoConnection*”, e sua finalidade básica é ter métodos que retornem instâncias da classe *Mongo*, da classe *Morphia*, e da classe *Datastore*.

Primeiramente, como pode ser visto na Figura 14, foram criadas três constantes do tipo *String*, com modificador privado, que serão utilizadas somente internamente nesta classe, para guardar o nome do banco, o *host* de acesso do banco (caminho aonde o banco é encontrado, no caso de banco local na mesma máquina, *localhost*) e a porta.

```
private static final String BANCO = "gpa";  
private static final String HOST = "localhost";  
private static final int PORT = 27017;
```

Figura 14 - Constantes da classe *MongoConnection*

Tendo as constantes, o primeiro método a ser implementado foi o “*getMongo*”, que retorna uma instancia de *Mongo* que é a principal classe de ligação do *MongoDB* com a linguagem Java. Primeiramente foi feito o método normalmente sem *static*, mas cada vez que o banco era requisitado, ele retornava uma nova instancia de *Mongo*, e abria uma nova conexão no banco de dados.

Como pode ser visto na Figura 15, para resolver o problema de múltiplas conexões *Mongo* sendo abertas, foi criado um Objeto estático da classe *Mongo*, com modificador privado, e no método *getMongo*, foi implementado o padrão de projeto *singleton*, que garante que o método retorne sempre a mesma instancia de *Mongo*, esta que fica sempre na memória do servidor, pelo fato do objeto ter sido declarado estático. Foram criados Objetos semelhantes para a instancia de *Morphia* e para a instancia de *Datastore* também, pois ambos terão métodos com *singleton* com a mesma finalidades semelhantes a do método *getMongo*.

```
private static Mongo mongo;
private static Morphia morphia;
private static Datastore datastore;
```

Figura 15 - Objetos estáticos utilizados nos métodos *singleton*

```
/**
 * Retorna uma instancia de Mongo. Caso nao tenha nenhuma, cria uma.
 */
private static Mongo getMongo()
{
    if(mongo == null)
    {
        try
        {
            mongo = new Mongo(HOST, PORT);
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
    return mongo;
}
```

Figura 16 - Método *getMongo*

A Figura 16, ilustra como ficou a implementação do método `getMongo`, o qual ficou com modificador `static` por que o objeto `Mongo` foi declarado estático. O retorno do método é um objeto `Mongo` de mesmo tipo do objeto `Mongo` declarado global na classe.

Primeiramente no método, verifica-se se o objeto `Mongo` global esta nulo, e se estiver nulo cria-se uma nova instancia de `Mongo` passando para o construtor o `host` e a porta, ambos declarados nas constantes. Esta criação esta dentro de um bloco `try-catch`, assim caso ocorra uma `exception` (banco não iniciado, por exemplo) o sistema dispara ela para possível tratamento ou simplesmente mostrar a mensagem na parte cliente.

Após a criação a primeira vez, a variável global `Mongo` passa a guardar esta instancia do banco, e todas as chamadas subseqüentes ao método `getMongo` retornarão aquela mesma instancia, não chamando novamente o construtor da classe `Mongo` durante esta execução do sistema, caso o servidor seja parado, obviamente o objeto `Mongo` será criado novamente.

```
/**
 * Retorna uma instancia do Morphia. Caso nao tenha nenhuma cria uma, e mapea as classes.
 */
private static Morphia getMorphia()
{
    if(morphia == null)
    {
        morphia = new Morphia();
        morphia.map(UsuarioPOJO.class)
                .map(StakeholderPOJO.class)
                .map(ProjetoPOJO.class)
                .map(EncaminhamentoPOJO.class)
                .map(RequisitoPOJO.class)
                .map(ProdutoPOJO.class);
    }
    return morphia;
}
```

Figura 17 - Método `getMorphia`

Como apresentado na Figura 17, o método `getMorphia` também tem implementação estática, também implementa o padrão `singleton` e também é ligado diretamente a um objeto estático global com seu tipo sendo o mesmo tipo de retorno do método. Quando o objeto `Morphia` ainda não foi criado, ele cria uma instancia de `Morphia` chamando o construtor da classe sem passar nenhum parâmetro.

Em seguida, é feito o mapeamento dos *POJOS*, que são as classes com o mapeamento do *Morphia*, indicando para o *framework Morphia*, quais serão as classes mapeadas pelo *MongoDB*.

```
/**
 * Retorna o datastore. Caso nao tenha nenhum cria um.
 */
public static Datastore getDatastore()
{
    if(datastore == null)
    {
        datastore = getMorphia().createDatastore(getMongo(), BANCO);
        datastore.ensureIndexes();
    }
    return datastore;
}
```

Figura 18 - Método `getDatastore`

Os bancos de dado não relacionais atualmente têm ganhado vários adeptos e seguidores, devido à escalabilidade e facilidade de uso proporcionado pela grande maioria dos bancos de dados não relacionais. Paralelamente a isto, na área de gerenciamento de projetos, a metodologia ágil Scrum vem se destacado, devido à facilidade e controle do tempo gasto mais perto possíveis do real.

Este trabalho tem intuito de apresentar um estudo sobre o Scrum e sobre a implementação de uma aplicação para gerenciamento de etapas do Scrum, visando à apresentação da implementação do banco de dados *MongoDB*, com utilização de alguns *frameworks*, como o Google Guice e o *Morphia*.

O método `getDatastore` é de extrema importância para o sistema, pois ele será usado nos construtores de todos os *DAOs* implementados no sistema, e levará para eles a conexão com o banco de dados. Apresentado na Figura 18, o método `getDatastore` é estático, tem um Objeto global estático assim como *Mongo* e *Morphia*, e é responsável por retornar um objeto do tipo *Datastore*.

Quando o objeto *Datastore* global esta nulo, o *Datastore* é criado pela chamada do método *createDatastore* da classe *Morphia*, passando como parâmetro a instancia de *Mongo*, e o nome do banco. Como já foram implementados métodos para pegar instancias de *Mongo* e *Morphia*, o trabalho para o método *getDatastore* fica muito facilitado, e o nome do banco também tem uma constante guardando o nome do banco, assim não há perigo de mudar “sem querer” o nome do banco.

Após o objeto *Datastore* criado, é chamado o método *ensureIndexes*, que adiciona os índices anotados nas classes *POJO*, no *MongoDB*, e deixa pronto para as buscas por índice(Que são feitos automaticamente pelo *MongoDB* e pelo *Morphia*).

4.1.3 Classes DAO

Tendo implementada a classe *MongoConnection*, agora o sistema tem comunicação com o banco de dados, mas a parte cliente do sistema ainda não tem ligação direta com o banco de dados. Para isto, deve ser implementadas classes *DAO*, para servir de ligação entre o banco de dados e as classes que se comunicam com a parte cliente da aplicação.

Para a criação dos *DAOs*, primeiro foi criada uma interface chamada *IDao*, que serve como base para implementação de todos os outros *DAOs*. Como pode ser visto na Figura 19, a interface *IDao* recebe dois *generics*, T e K, representando o objeto *Value Object* do *DAO*, e o objeto *POJO* do *DAO* respectivamente. Por Exemplo, para o *DaoProjeto* a implementação de *IDao* usaria no lugar de T a classe *Projeto*, e no lugar de K a classe *ProjetoPOJO*.

```

* Interface para os Daos. Aqui ficam descritos todos os metodos comuns para todos os Daos.
public interface IDao<T, K>
{
    * Salva o objeto e retorna o objeto salvo ja com id.
    T salvar(T t);

    * Exclui o objeto e retorna um resultado. Obrigatoriamente o objeto deve ter id.
    boolean excluir(T t);

    * Busca uma lista com todos os objetos do banco sem nenhum limite ou criterio.
    List<T> buscarTodos();

    * Busca um objeto <T> de acordo com seu id.
    T buscar(String id);

    * Converte uma lista de resultados de um metodo find() para uma lista de valueObjects.
    List<T> toValueObject(QueryResults<K> resultadoBusca);
}

```

Figura 19 - Interface IDao

Os métodos implementados na interface IDao serão comuns a todas as classes que implementarem a interface IDao, por isso os métodos que todas as classes DAO usarão foram implementados no IDao:

- salvar – Salva o objeto T recebido como parâmetro, e retorna o próprio objeto já salvo com Id preenchido.
- excluir – Exclui da coleção correspondente o documento com Id igual ao Id do objeto T passado como parâmetro. Retorna um *boolean* dizendo se a exclusão foi bem sucedida.
- buscarTodos – Busca na coleção todos os objetos, sem passar nenhuma restrição ou filtro para a busca. Retorna uma lista de Objetos T.
- buscar – Busca na coleção o objeto correspondente ao id passado como parâmetro. Retorna um objeto T, ou null caso não encontre nenhum objeto.
- toValueObject – Método auxiliar usado somente dentro da própria classe DAO. Recebe um objeto *QueryResults*, que é o retorno de um método *find* do *BasicDAO* do *Morphia*, método este que vai ser explicado mais detalhadamente adiante. A função do *toValueObject* é converter o resultado da busca, que vem em forma de objetos *POJO* para *Value Objects*, e retornar.

Tendo a interface IDao implementada, os DAOs já poderiam facilmente ser criados, mas há DAOs que precisariam de mais métodos além destes, como buscas

específicas por campos que não existem em outros *DAOs*. Por exemplo, o *DaoStakeholder* tem um método `buscaByUsuario`, e o *DaoStakeholder* é a única classe no sistema que precisa ter este método implementado, por este motivo não seria interessante o `IDao` ter este método declarado. Porém deixar sem declaração um método de busca é inseguro e não seria a melhor solução para o problema, assim para resolver este problema outras interfaces precisam ser criadas, para conter os métodos específicos de cada *DAO*.

Cada *DAO* criado no sistema terá obrigatoriamente, uma *interface* própria, *interface* esta que herdará a *interface* `IDao` com os métodos que todos os *DAOs* precisam implementar. Para exemplo será usado o `IDaoProjeto`, *interface* esta que é usada pra controle do *DaoProjeto*.

```
public interface IDaoProjeto extends IDao<Projeto, ProjetoPOJO>
{
    List<Projeto> buscarLike(String parametro, Usuario usuario);

    List<ProjetoStakeholderDTO> buscarByUsuario(Usuario usuario);

    Projeto carregarRequisitos(Projeto projeto);
}
```

Figura 20 - Interface `IDaoProjeto`

Como pode ser visto na Figura 20, a *interface* `IDaoProjeto` é a *interface* que vai servir como ligação entre a *interface* `IDao` e a classe *DaoProjeto*. Os métodos específicos para o *DaoProjeto* são declarados nesta *interface*, métodos estes que só poderão ser implementados na classe que implementar `IDaoProjeto`, no caso, a classe *DaoProjeto*.

A *API* do *Morphia Framework* tem uma classe chamada *BasicDAO* que dá uma base pré-pronta para a construção de *DAOs* no sistema. A classe *BasicDAO* recebe como parâmetro a classe na qual o *Morphia* vai usar para salvar os dados no banco, ou seja, a classe *POJO* correspondente ao objeto, e também a classe que servirá como *Id* daquele documento, no caso *ObjectID*.

```

public class DaoProjeto extends BasicDAO<ProjetoPOJO, ObjectId> implements IDaoProjeto
{
    @Inject
    public DaoProjeto()
    {
        super(MongoConnection.getDatastore());
    }

    public DaoProjeto(Datastore ds)
    {
        super(ds);
    }

    //metodos override da interface ...
}

```

Figura 21 – DaoProjeto

Todas as classes *DAO* do sistema, obrigatoriamente irão herdar de *BasicDAO*, e irão implementar a *interface* *IDao* correspondente. Como apresentado na Figura 21, a classe *DaoProjeto* herda de *BasicDAO* com as classes certas pré-configuradas nos *generics*, e implementa a *interface* *IDaoProjeto*, que por sua vez herda de *IDao*, assim o *DaoProjeto* precisa implementar os métodos tanto de *IDao* quanto de *IDaoProjeto*.

Outro pré-requisito para a classe é um construtor com a anotação *@Inject*, que é usado pelo Google Guice para indicar que esta classe será injetada na classe ou interface pré-configurada pelo Google Guice, assunto este que será tratado mais adiante. Este construtor deve chamar o construtor da superclasse através do comando *super*, passando como parâmetro o *Datastore* do banco de dados, que será pego da classe *MongoConnection*.

Opcionalmente, pode ser implementado outro construtor sem injeção de dependência, de acordo com a necessidade, mas obrigatoriamente o construtor com *@Inject* deve ser implementado, na atual arquitetura do sistema.

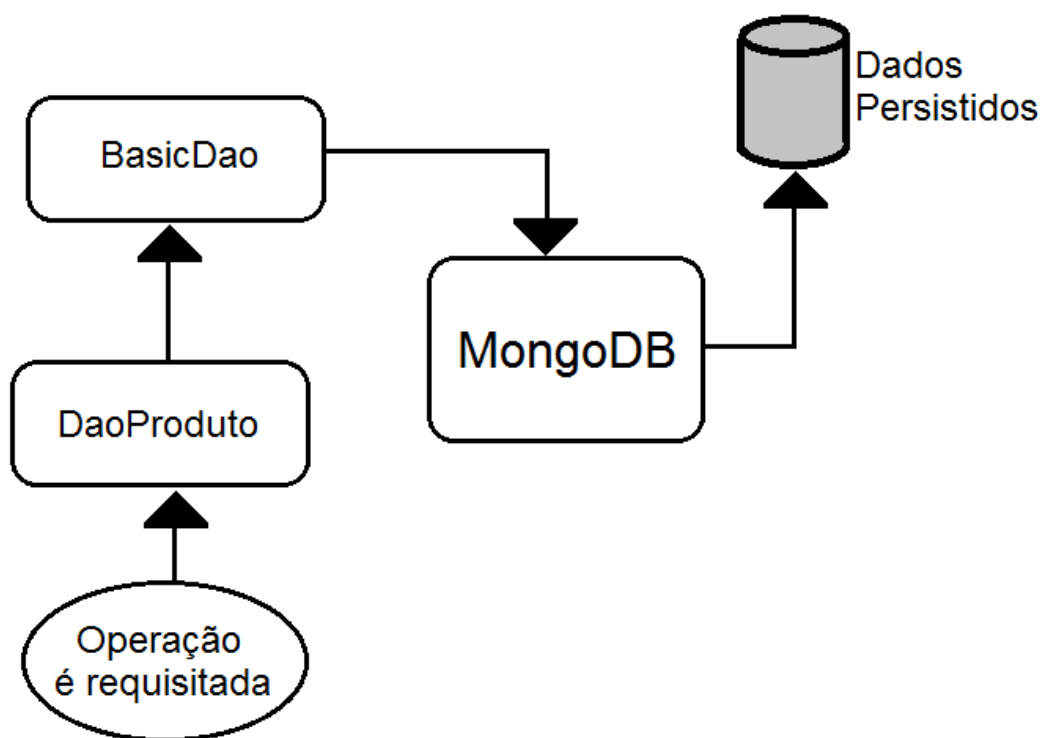


Figura 22 - Diagrama sobre o funcionamento básico dos DAOs

A Figura 22 apresenta o funcionamento básico de um *DAO* no sistema. Primeiramente, a operação é requisitada, pela parte cliente da aplicação ou mesmo por outro *DAO*, então é passada a classe *DAO* específica, por exemplo, se uma tela requisita a operação de salvar produto, a ordem de chamar o método salvar vem da parte cliente (injetada pelo *framework* Google Guice), a operação chega ao *DAO* específico, no caso *DaoProduto*.

Ao chegar ao *DaoProduto*, a operação é processada pelo método específico, no caso o método salvar.

O método salvar faz alguns tratamentos e manda para a classe pai, o *BasicDao*, e o *BasicDao* se relaciona internamente com o *MongoDB* e gera as *Querys* necessárias à execução da operação.

4.1.4 Implementação de o Método Salvar

O método salvar é declarado na *interface* `IDao` e é comum a todos os *DAOs* implementados no sistema.

```

39 @Override
40 public Projeto salvar(Projeto t)
41 {
42     ProjetoPOJO projPojo = new ProjetoPOJO(t);
43     Key<ProjetoPOJO> key = this.save(projPojo);
44     if(key == null)
45     {
46         return null;
47     }
48     System.out.println("Salvo com id " + key.toString());
49
50     projPojo.setId(new ObjectId(key.getId().toString()));
51     return projPojo.getProjeto();
52 }
53

```

Figura 23 - Método salvar

A Figura 23, mostra a implementação de o método salvar do `DaoProjeto`. A Anotação `@Override`, encontrada na linha 39 da Figura 23 é obrigatória, pois indica ao compilador que este método é implementação do método declarado na interface `IDao`. Todos os métodos Salvar do sistema são exatamente iguais, só mudando as classes específicas, por exemplo, o `DaoProjeto` salva objetos instancias da classe `ProjetoPOJO` enquanto o `DaoProduto` salva objetos instancia da classe `ProdutoPOJO`.

A primeira coisa a ser feita no método salvar é criar um objeto *POJO*, pois o método recebe um objeto *Value Object*, mas o banco salva somente objetos *POJO*. Para isto só é necessário a criação do objeto *POJO*, e passar ao construtor do objeto *POJO* o objeto *Value Object* recebido por parâmetro, como mostrado na linha 42 da Figura 23.

Tendo o objeto *POJO*, declara-se um objeto instancia da classe *Key* do *Morphia*, declaração esta que precisa que seja indicada qual classe esta *Key* pertence e este objeto recebe o resultado da chamada do método *save* do *BasicDAO*, passando como parâmetro o objeto *POJO*. Este processo, que pode ser visto na linha 43 da Figura 23, é o código que de fato salva o objeto na coleção correspondente, e caso salve com sucesso, o método *save* retorna uma instancia de *Key*, contendo o id do objeto salvo no banco neste respectivo documento.

Após salvo, é verificado se a *Key* esta nula, e caso esteja nula, indica que ocorreu algum erro ao salvar, então retorna *null* na operação inteira. Caso tenha sido salvo com sucesso e não retorne, é adicionado ao objeto *POJO* o Id do objeto que foi salvo, pela chamada do método *setId*, passando como parâmetro um *new*

ObjectID com o valor do id da *Key*, já convertido para *String*, como pode ser visto na linha 50 da Figura 23.

Com o objeto *POJO* já salvo e com *Id* já preenchido, retorna-se o objeto *POJO* chamando o método *getObjeto*, no caso da Figura 23, chamando-se o *getProjeto*(este nome varia de acordo com o *DAO*), este processo pode ser visto na linha 51 da Figura 23.

4.1.5 Implementação de o Método Excluir

A implementação de o método excluir, a exemplo da implementação do método salvar, também é similar em todos os outros DAOs do sistema.

```
54 @Override
55 public boolean excluir(Projeto t)
56 {
57     try
58     {
59         this.deleteById(new ProjetoPOJO(t).getId());
60         return true;
61     }
62     catch(Exception e)
63     {
64         e.printStackTrace();
65         return false;
66     }
67 }
```

Figura 24 - Método `excluir`

Percebe-se na Figura 24, que a implementação do método `excluir` é relativamente simples em comparação a de o método `salvar`. Na linha 57 da Figura 24, foi aberto um bloco `try-catch`, para capturar possíveis `exceptions` que podem ser geradas por este método, como por exemplo, se um objeto com `Id` nulo for recebido como parâmetro pelo método `excluir`. Na linha 59 da Figura 24, é onde a exclusão é de fato realizada, pela chamada do método `deleteById` do `BasicDAO`, passando como parâmetro o `id` do objeto recebido como parâmetro, com tipo `ObjectId`.

Caso não ocorram `exceptions` durante a exclusão, como visto na linha 60 da Figura 24 é retornado o valor `boolean true`, indicando que a exclusão foi realizada com sucesso, e caso ocorra alguma `exception`, ela vai ser capturada pelo `try-catch`, que vai lançar a `exception` com a chamada do método `printStackTrace`, e logo após será retornado o valor `boolean false`, indicando que a exclusão não foi realizada com sucesso.

4.1.6 Implementação do método *Buscar*, *BuscarTodos* e *ToValueObject*

Uma das grandes vantagens de trabalhar com *MongoDB*, é a facilidade com que buscas podem ser feitas, devido ao modo que os objetos são salvos no banco de dados (como documentos *JSON*). Além deste fato, o *Morphia* facilita muito a criação de buscas com métodos específicos para buscar e criação de *Querys MongoDB*.

```

75 @Override
76 public Projeto buscar(String id)
77 {
78     ProjetoPOJO projPojo = this.findOne("id", new ObjectId(id));
79     if(projPojo == null)
80     {
81         return null;
82     }
83     return projPojo.getProjeto();
84 }

```

Figura 25 - Método *buscar*

Apresentado na Figura 25, o método *buscar* tem como função básica, retornar um objeto *Value Object*, a partir de um *id* de tipo *String*, recebido como parâmetro pelo método. A implementação do método *buscar* é relativamente simples, só sendo necessário chamar o método *findOne*, filtrando pelo campo *id* que foi declarado no *POJO* com anotação *@Id*, e criando um objeto *ObjectId* e passando como parâmetro ao seu construtor o *id* de tipo *String* recebido por parâmetro. Este processo pode ser visto na linha 78 da Figura 25.

```

69 @Override
70 public List<Projeto> buscarTodos()
71 {
72     return toValueObject(this.find());
73 }

```

Figura 26 - Método *Buscartodos*

O método *buscarTodos* se resume a uma única linha de código. A única implementação necessária para se buscar todos os registros de uma *collection*, é chamar o método *toValueObject* (método este que não é herdado do *BasicDAO*, e sim implementado em todo *DAO*), passando ao método o retorno da chamada da função *find* do *BasicDAO*. Caso a função *find* seja chamada sem receber nenhum

parâmetro, ela busca automaticamente por todos os dados da *collection* correspondente a ligação com o *BasicDAO*. Já o método *toValueObject* é responsável por um objeto *QueryResult* (retorno da chamada de um método *find*) em uma lista de *ValueObject*.

```

158 @Override
159 public List<Projeto> toValueObject(QueryResults<ProjetoPOJO> resultadoBusca)
160 {
161     List<Projeto> retorno = new ArrayList<Projeto>();
162     for(ProjetoPOJO projPojo : resultadoBusca.asList())
163     {
164         retorno.add(projPojo.getProjeto());
165     }
166     return retorno;
167 }

```

Figura 27 - Método *toValueObject*

O método *toValueObject* tem implementações bem similares em todos os *DAOs*, e é chamado para o mesmo propósito em todos eles. Como pode ser visto na Figura 27, o método recebe um objeto *QueryResults* como parâmetro, vindo do retorno de um método *find*, assim obrigatoriamente, o método *toValueObject* sempre vai ser chamado após a chamada de um método *find*.

Basicamente, a implementação do método *toValueObject* consistem em fazer uma iteração “for” no *QueryResults*(possível pela chamada do método *asList* da classe *QueryResults*), e adicionando a um *Array* do *Value Object* correspondente ao *DAO*. Ao fim da iteração é retornado o *Array*, contendo os valores que estavam dentro do objeto *QueryResults*. E esta iteração somente é necessária, pois os registros de *QueryResults* estão com tipo *POJO*, mas eu preciso deles com tipo de *Value Objects*.

4.2 INJEÇÃO DE DEPENDÊNCIA – GUICE

Com os *DAOs* prontos, a classe *MongoConnection* criado, os *valueObjects* criados e os *POJOs* implementados e mapeados, ainda fica pendente a ligação entre a parte cliente da aplicação (*GWT*) e a parte de persistência da aplicação. Para este fim, foi implementada injeção de dependência com uso do *framework* Google Guice.

A implementação de Google Guice para injeção de dependência no sistema, foi com o foco de eliminar a responsabilidade do programador em tratar a ligação do cliente com a persistência, ao invés disto, delegar a responsabilidade ao *framework* de realizar a operação e somente esperar o resultado, eliminando vários *Factories* (padrão de projeto *Factory*), que conseqüentemente seriam implementados para este fim.

Para iniciar a utilização do Google Guice na aplicação, inicialmente deve-se adicionar os *JARs* do Google Guice no projeto da aplicação, e configurar seu *BuildPath*. Depois de configurado o ambiente com os *JARs*, cria-se um pacote para a injeção de dependência, de preferência no lado servidor da aplicação (no caso do sistema proposto, obrigatoriamente no lado servidor da aplicação, pois GWT não permite a serialização das classes do Google Guice no lado cliente).

Depois de criado os pacotes, foram criadas classes para configurar as injeções posteriormente feitas. No sistema, o Google Guice foi utilizado para injetar os *DAOs* dentro de cada *ActionHandler* do sistema (classe responsável por tratar as requisições vindas da parte cliente da aplicação), esta classe foi criada com nome *DAOModule*.

```

17 public class DAOModule extends AbstractModule
18 {
19
20     @Override
21     protected void configure()
22     {
23         bind(IDaoUsuario.class).to(DaoUsuario.class);
24         bind(IDaoStakeholder.class).to(DaoStakeholder.class);
25         bind(IDaoEncaminhamento.class).to(DaoEncaminhamento.class);
26         bind(IDaoRequisito.class).to(DaoRequisito.class);
27         bind(IDaoProjeto.class).to(DaoProjeto.class);
28         bind(IDaoProduto.class).to(DaoProduto.class);
29     }
30 }
31 }
32 }
33 }

```

Figura 28 - Classe *DAOModule*

Como pode ser visto na Figura 28 a classe *DAOModule* herda de *AbstractModule*, que é uma classe abstrata e torna obrigatório a implementação do método *configure* na classe *DAOModule*, como pode ser visto na linha 21 da Figura 28.

O método *configure* é chamado automaticamente pelo Google Guice, sendo responsabilidade do programador somente implementá-lo corretamente, mas não sendo responsável por chamá-lo. No método *configure*, tudo que é feito é chamar o

método *bind*, passando como parâmetro a *interface* específica do *DAO*, logo depois chamando o método *to*, passando como parâmetro o próprio *DAO* da interface que foi passada no método *bind*, desta maneira “ligando” a *interface* `IDaoObjeto` com o `DaoObjeto`. Estas ligações podem ser vistas nas linhas 23 a 28 da Figura 28.

Com o ambiente do Google Guice previamente configurado, e a correta implementação do `DAOModule`, a injeção de dependência pode ser feita nas classes que necessitam a ligação direta com os *DAOs* do sistema. Essas classes, são as classes chamadas de *ActionHandlers*.

Os *ActionHandlers* são classes localizadas na parte Servidor da aplicação, que têm a responsabilidade de receber chamadas da parte Cliente da aplicação, executar a operação necessária e enviar uma resposta a parte Cliente da aplicação, respostas estas que utilizam o padrão *DTO* (*Data Transfer Object*), para retornar os dados.

Na parte Cliente da aplicação dois tipos de classes são implementadas para trabalhar em conjunto com os *ActionHandlers*: as classes *Actions* e as classes *Result*.

As classes *Action*, são as responsáveis por guardar informações sobre os dados necessários para a realização da operação. Por exemplo: caso a operação for uma operação de salvar um objeto no banco, a classe “`SalvarObjetoAction`” seria responsável por guardar o objeto a ser salvo, para a injeção de dependência da parte Cliente injetar esta dependência ao *ActionHandler* correspondente⁷.

As classes *Result*, são *DTOs* responsáveis por guardar informações sobre os dados que retornarão da parte servidor da aplicação, até que alguma classe da parte Cliente da aplicação faça uso destas informações.

Abstraindo um pouco as informações sobre *Action*, *ActionHandler* e *Result*, pode-se dizer que a classe *Action* guarda a chamada da operação, a classe *ActionHandler* executa a operação e retorna uma classe *Result* com o resultado da operação, que será usado pela parte cliente da aplicação.

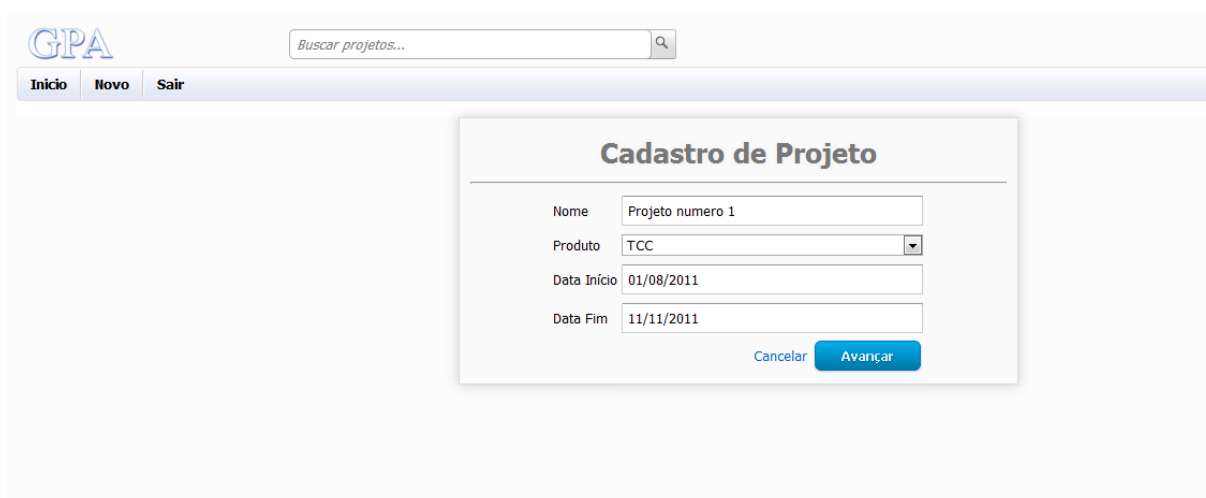
⁷ Esta injeção de dependência não é feita pelo Google Guice, mas sim pelo `gwtPlatform`. Mais detalhes podem ser vistos no Trabalho de conclusão de curso do Carlos Alexandre Becker, que trata sobre a parte Cliente desta mesma aplicação.

Mais detalhes sobre as configurações do Google Guice, podem ser encontrados no site oficial do Google Guice⁸.

4.3 RESULTADOS ALCANÇADOS

Os resultados alcançados foram extremamente satisfatórios, pois o sistema GPA executa as operações em poucos instantes, tanto rodando em servidor local quanto executando um servidor remoto. Não foram feitos testes mais aprofundados sobre a velocidade da aplicação, mas ao usar a aplicação percebe-se visualmente que a velocidade é um grande ponto forte.

Com o uso de vários padrões de projeto, e dos *frameworks* Google Guice para injeção de dependência, Morphia para a persistência, MongoDB como banco de dados, a manutenção de código fica muito facilitada, pois boas práticas de programação eliminam muitos dos possíveis problemas que podem ser gerados.



A imagem mostra a interface web do sistema GPA. No topo, há o logotipo 'GPA' e uma barra de busca com o texto 'Buscar projetos...'. Abaixo, há uma barra de navegação com os links 'Início', 'Novo' e 'Sair'. O conteúdo principal é um formulário modal intitulado 'Cadastro de Projeto'. O formulário possui os seguintes campos:

- Nome: Projeto numero 1
- Produto: TCC (selecionado em uma lista suspensa)
- Data Início: 01/08/2011
- Data Fim: 11/11/2011

Na base do formulário, há dois botões: 'Cancelar' e 'Avançar'.

Figura 29 - Tela de cadastro de Projeto

Como pode ser visto na Figura 29, a tela de cadastro de projeto foi implementada em forma de “*wizard*”, possibilitando ao usuário cadastrar seu projeto aos poucos, sem preencher todos os dados de uma única vez. Inicialmente o usuário preenche os valores de “Nome do projeto”, seleciona o “Produto” de uma lista, e seleciona a data de cadastro, e a data prevista para termino do projeto. Ao clicar em avançar o usuário é direcionado para a tela responsável por cadastrar os requisitos do projeto.

⁸ O site oficial do Google Guice pode ser encontrado em <<http://code.google.com/p/google-guice/>>

The screenshot shows the GPA system interface. At the top, there is a search bar labeled 'Buscar projetos...' and navigation buttons for 'Início', 'Novo', and 'Sair'. The main content area is titled 'Adicionar Requisitos ao Projeto'. On the left, there is a 'Requisitos' section with a list of existing requirements, including 'Requisito 1 (10 hrs)'. The main form area contains the following fields and controls:

- Título:** A text input field containing 'Requisito 2'.
- Prioridade:** A dropdown menu set to 'Alta'.
- Tempo estimado:** A text input field containing '30' followed by the unit 'Horas'.
- Formatação:** A rich text editor toolbar with buttons for Bold (B), Italic (I), Underline (U), Text Color (ABC), Background Color (x), and other formatting options.
- Descrição:** A large text area containing the placeholder text 'O requisito 2....'.
- Controles de Projeto:** At the bottom left, a status indicator shows 'Projeto possui 10 hrs' with minus and plus buttons.
- Botões de Ação:** At the bottom right, there are buttons for 'Cancelar', 'Voltar', and 'Avançar', along with a 'Salvar Requisito' button.

Figura 30 - Tela de cadastro de Requisitos do Projeto

Como pode ser visto na Figura 30, a tela de cadastro de requisitos do projeto conta com os campos:

- Título – Título do requisito. Será o nome mostrado no *Post-it* no quadro de scrum.
- Prioridade – Prioridade de desenvolvimento do requisito. Pode ser Alta, Média ou Baixa. De acordo com a prioridade a cor da borda do Post-it da tela principal mudará, possibilitando assim o rápido entendimento do requisito, sem a necessidade de abrir detalhadamente.
- Tempo estimado – Tempo estimado para o termino do requisito.
- Descrição – Campo com a descrição detalhada do necessário para a conclusão do requisito.

A esquerda nesta mesma tela, tem uma lista dos requisitos já adicionados no sistema, e também dois botões, um de “+” usado para adicionar novos requisitos, e outro de “-” usado para remover o requisito selecionado na lista do projeto.

Ao clicar no botão avançar, é apresentada ao usuário a tela de seleção de *Stakeholders* para o projeto.

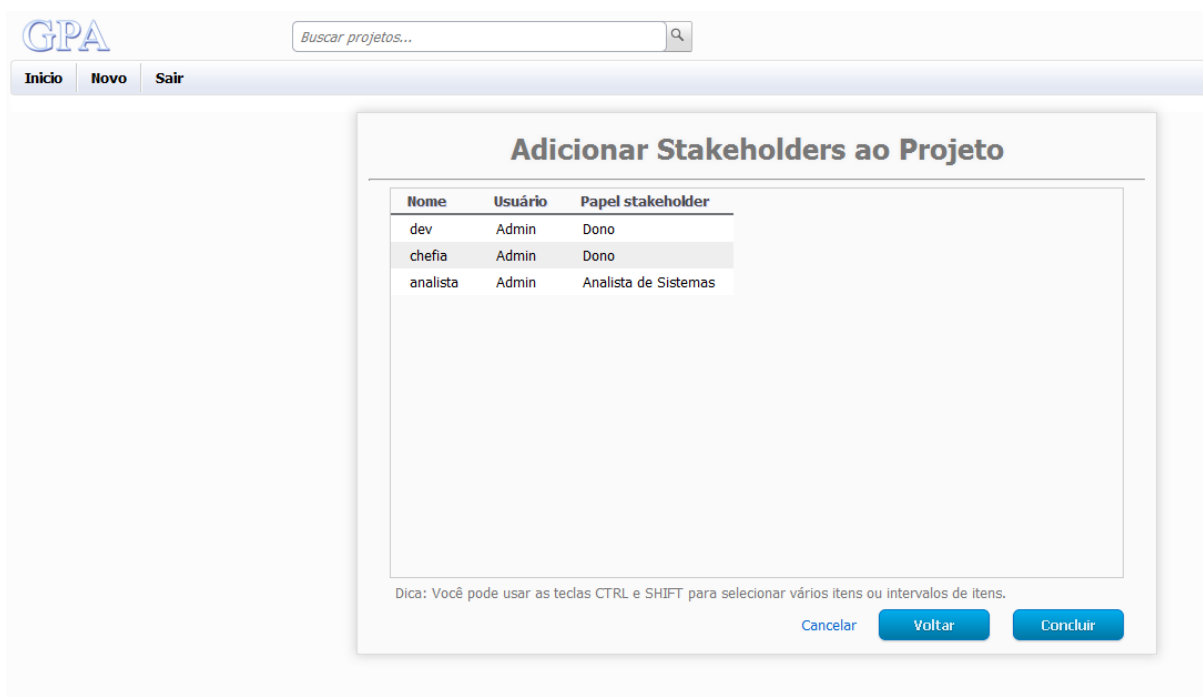


Figura 31 - Tela de seleção de *stakeholders* para o projeto

Na Figura 31 é apresentada a seleção de *stakeholders* para o projeto, a seleção é feita simplesmente clicando sobre o *stakeholder*, todos os *stakeholders* selecionados serão usados no projeto. Ao clicar em concluir, o projeto é salvo e passa a aparecer na lista de projetos do usuário logado na tela principal do sistema.



Figura 32 - Quadro de Scrum na tela principal do sistema

Como visto na Figura 32, a tela principal do sistema apresenta a lista de projetos do usuário logado, uma barra de tarefas com a possibilidade de criação de novos usuários, Stakeholders, Produtos e Projetos, um botão que volta a tela principal e o botão sair para deslogar da aplicação.

Ao lado da lista de projetos do usuário, o quadro de scrum com os *post-its* faz o controle do fluxo dos requisitos. Cada requisito cadastrado no projeto é um *post-it* na tela, e sua posição é escolhida de acordo com último encaminhamento cadastrado para aquele requisito. A cor da borda do *post-it* muda de acordo com a prioridade do requisito :

- Borda Vermelha – o requisito tem prioridade alta de desenvolvimento;
- Borda Laranja – o requisito tem prioridade média de desenvolvimento;
- Borda Verde – o requisito tem prioridade baixa de desenvolvimento.

Os encaminhamentos só podem ser feito para o próximo Status Requisito, ou para o Anterior, ou seja, não é possível passar um requisito de Aguardando para “Em Testes” diretamente, obrigando o usuário a seguir o fluxo pré-definido de execução.

5 CONSIDERAÇÕES FINAIS

Este capítulo apresenta a conclusão de todo o trabalho realizado, citando os resultados que eram esperados e os resultados que foram obtidos, sobre cada uma das tecnologias. Também cita quais são as futuras implementações do projeto desenvolvido neste trabalho.

5.1 CONCLUSÃO

A implementação de um banco de dados *NoSQL*, em uma aplicação real, tem grandes vantagens devido à velocidade e fácil aplicação e interligação com variados *frameworks* e até linguagens de programação existentes, mas o fato da maioria dos banco de dados *NoSQL* não serem *ACID* deixam duvidas sobre sua verdadeira capacidade de segurança dos dados.

O MongoDB não se difere neste ponto, pois também tem mais propriedades de *BASE* do que propriedades *ACID*. Mas em compensação, a escalabilidade vertical e horizontal proporcionada pelo MongoDB, consegue totalmente se sobrepor à dificuldade quanto à segurança. Vale ressaltar que os sistemas *NoSQL* não são inseguros o tempo inteiro, e sim, tem picos de momentos “desprotegidos”.

Em uma aplicação com missão do cunho de controlar projetos utilizando esta metodologia ágil, possivelmente pode não ter problemas com o banco de dados, com segurança dos dados trafegados, e deve ser rápida para não tornar inviável seu uso. Neste ponto o *framework* Morphia mostrou-se muito útil, pois além de facilitar o desenvolvimento para o MongoDB, ele dispõe de mecanismos que minimizam a falta das propriedades *ACID*, como controle de versionamento, de maneira a deixar o trabalho com o MongoDB praticamente tão seguro quanto um banco de dados relacional, mas com o diferencial que é mais rápido e livre de burocracias quanto a modificações na base de dados.

O uso de injeção de dependência com Google Guice mostrou-se bem eficiente e realmente elimina várias instanciações desnecessárias das classes *DAO*, o que potencialmente aumentaria os riscos de problemas com as classes de persistência. Apesar disso, sua configuração é um pouco complicada de entender no início, mas depois de feita dificilmente é modificada.

5.2 TRABALHOS FUTUROS/CONTINUAÇÃO DO TRABALHO

O projeto GPA (Gerenciador de Projetos Ágeis), como foi nomeada a aplicação desenvolvida neste trabalho, teve sua primeira versão implementada com objetivo que ficasse rápida e utilizável ao gerenciamento de um projeto real. Vários aspectos importantes serão futuramente desenvolvidos, como a geração do gráfico de *sprint/backlog*, outros Estados de Requisito não foram desenvolvidos deixando somente quatro estados fixos, tipos de *Stakeholder* ficaram fixos, o *upload* de anexos para o projeto, requisitos e encaminhamentos. Também será melhorada a usabilidade do sistema em geral, e deixar mais perto possível de um sistema que atenda o real controle de *Scrum*.

REFERÊNCIAS BIBLIOGRÁFICAS

BEZERRA, Eduardo, **Princípios de Análise e Projeto de Sistemas com UML**. Editora Campus; 2007.

CodeFutures.com, <<http://www.codefutures.com/java-dao/>>, Acesso em 06 Set 2011.

Google Guice <<http://code.google.com/p/google-guice/>>, Acesso em 14 out 2011.

Fayad, Mohamed e Schmidt, C. Douglas, **Magazine Communication of the ACM**. 1997.

Free Software Foundation, <<http://www.fsf.org/>>, Acesso em 10 set 2011.

Fowler, Martin <<http://www.martinfowler.com/articles/injection.html>>, Acesso em 17 out 2011.

KNIBERG, Henrik. **Scrum e XP direto das Trincheiras: como fazemos SCRUM**. C4Media Inc, 2004. Distribuído por InfoQueue em <<http://infoq.com/br/minibooks/scrum-xp-fromthe-trenches>>. Acesso em 23 de Novembro de 2011.

Lemay, Laura. Perkins, L. Charles. **Teach Yourself JAVA in 21 Days**. Editora Sams.net. 1999.

Machado, Magno <<http://blog.magnomachado.com.br/inversao-de-controle-ioc-e-injecao-de-dependencias-di/>>, Acesso em 17 out 2011.

MongoDB.org <<http://www.mongodb.org>>, Acesso em 16/10/2011>.

Mountain Goat Software <<http://mountaingoatsoftware.com/scrum>>, Acesso em 03 nov 2011.

Morphia <<http://code.google.com/p/morphia/>>, Acesso em 17 out 2011.

Porcelli, Alexandre, **Revista Java Magazine**. Edição 86. Editora DevMedia; 2010.

Silva, Lúcia da. Edna; Menezes. M. Estera. **Metodologia da pesquisa e Elaboração de Dissertação**. 3ª Edição. Ufsc.

SCHWABER, Ken. **Scrum Guide**. 2011 Disponível em <<http://www.scrum.org/storage/scrumguides/Scrum%20Guide%20-%202011.>> Acesso em 14 out 2011.

Strozzi.it <http://www.strozzi.it/cgi-bin/CSA/tw7/l/en_US/nosql>, Acesso em 17 out 2011.

VANBRABANT, Robbie; **Google Guice: Agile Lightweight Dependency Injection Framework (Firstpress)**, Apress, 2008.

VISUAL PARADIGM, **Company**. 200-. Disponível em <<http://www.visual-paradigm.com/aboutus/>>. Acesso em 09 out 2011.