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

CARLOS ALEXANDRO BECKER

DESENVOLVIMENTO DE APLICAÇÃO PARA O CONTROLE DE SCRUM UTILIZANDO GOOGLE WEB TOOLKIT

TRABALHO DE DIPLOMAÇÃO

MEDIANEIRA
2011

CARLOS ALEXANDRO BECKER

DESENVOLVIMENTO DE APLICAÇÃO PARA O CONTROLE DE SCRUM UTILIZANDO GOOGLE WEB TOOLKIT

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

Orientador: Prof Me. Fernando Schütz

MEDIANEIRA

2011

DEDICATÓRIA

Agradeço primeiramente a Deus, pois sem ele nada seria possível. Em seguida agradeço a minha família que sempre me incentivou a me dedicar, e também a minha noiva, Carine Daiane Meyer que vem sendo uma grande companheira, sempre me animando e me ajudando em diversos momentos. Não menos importante, agradeço a todos os meus amigos e professores, que sempre me apoiaram e ajudaram nos momentos de dificuldade.

“*Ser o homem mais rico do mundo no cemitério não me interessa. Ir para a cama dizendo que fizemos algo maravilhoso é o que importa.”*

Steve Jobs.

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 UTILIZANDO GOOGLE WEB TOOLKIT

Por

Carlos Alexandro Becker

Este Trabalho de Diplomação (TD) foi apresentado às 09:10 h do dia 22 de novembro de 2011 como requisito parcial para a obtenção do título de Tecnólogo no Curso Superior de Tecnologia em Manutenção Industrial, 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)

RESUMO

Becker, Carlos Alexandro. Desenvolvimento de Aplicação para o Controle de Scrum utilizando Google Web Toolkit. 2011. Trabalho de conclusão de curso (Tecnologia em Análise e Desenvolvimento de Sistemas), Universidade Tecnológica Federal do Paraná. Medianeira. 2011.

O *Google Web Toolkit* (GWT) é um *framework* de código aberto que permite que desenvolvedores criem aplicativos com tecnologia AJAX em linguagem de programação Java. O GWT enfatiza a reutilização de código e o uso de chamadas assíncronas. O GWT-Platform é um framework criado com a finalidade de aperfeiçoar o desenvolvimento de aplicações GWT, utilizando injeção de dependência, modelo MVP e diversos padrões de projeto. Este trabalho tem como objetivo demonstrar a utilização do GWT e do GWT-Platform para a construção de aplicações Web de forma ágil, concreta e simplificada, focando também no uso de injeção de dependências e a modularização do código.

Palavras-chave: GWT-Platform, Google Guice, Injeção de Dependência, Model-View-Present.

ABSTRACT

Becker, Carlos Alexandro. Desenvolvimento de Aplicação para o Controle de Scrum utilizando Google Web Toolkit. 2011. Trabalho de conclusão de curso (Tecnologia em Análise e Desenvolvimento de Sistemas), Universidade Tecnológica Federal do Paraná. Medianeira. 2011.

Google Web Toolkit is an open source framework that allows developers to easily create applications with AJAX technology in Java *programming language. GWT emphasizes code reuse and the use of asynchronous calls. GWT-Platform is a framework created in order to optimize the development of GWT applications, and especially to maintain solid standards for this using dependency injection, the MVP model and several design patterns. This work aims to demonstrate the use of GWT and GWT-Platform for building Web applications in an agile, practical and simplified way, also focusing on the use of Dependency Injection and modularization of code.*

Keywords: Scrum, GWT, GWT-Platform, Dependency Injection, MVP.

LISTA DE SIGLAS

AOP	*Aspect Oriented Programming*
AJAX	*Asynchronous* Javascript *and XML*
API	*Application Programming Interface*
CSS	*Cascading StyleSheet*
DAO	*Data Access Object*
DI	*Dependency Injection*
DAO	*Data Access Object*
Ecma	*European Computer Manufacturers Association*
GWT	*Google Web Toolkit*
GWTP	*GWT-Platform*
HTML	*HyperText Markup Language*
IoC	*Inversion of Control*
ISO	*International Organization for Standardization*
JSF	Java *Server Faces*
JSP	Java *Server Pages*
JSR	Java *Specification Request*
JSE	Java *Standard Edition*
JDK	Java *Development Kit*
JRE	Java *Runtime Environment*
MVC	*Model-View-Controller*

MVP	*Model-View-Presenter*
POJO	*Plain Old* Java *Object*
RPC	*Remote Procedure Call*
SDK	*Software Development Kit*
SVN	*Subversion*
URL	*Uniform Resource Locator*
UML	*Unified Modeling Language*
W3C	*World Wide Web Consortium*
XML	*Extensible Markup Language*

LISTA DE FIGURAS

LISTA DE QUADROS

SUMÁRIO

1 INTRODUÇÃO

A cada dia que passa, surge uma nova aplicação, seja para resolver algum problema ou para facilitar a execução de alguma tarefa para o usuário de um sistema informatizado. De modo geral, o objetivo da grande maioria delas é economizar o tempo do usuário final.

Paralelamente com a tecnologia, a Internet também evoluiu bastante. Os principais *browsers* lançam versões novas praticamente todo mês. Segundo PRESSMAN (2001), a *Web* deixou de ser orientada a documentos e passou a ser orientada a aplicações, podendo ter os mais variados objetivos. Isso tudo, está bastante associado à nova especificação para páginas da Internet, o HTML5 (*HyperText Markup Language*, versão 5). Segundo a W3C (2000), o HTML é uma linguagem para publicação de conteúdo (texto, imagem, vídeo, áudio e etc) na *Web*.

Porém, o HTML5, mesmo não estando pronto ainda, já é muito mais que isso. Segundo a W3C Brasil (2007), o HTML5, além de ser a nova versão do HTML4, e, ao contrário desta, possui várias ferramentas para que o CSS (*Cascading StyleSheet*) e o JavaScript possam fazer seu trabalho da melhor forma possível. Além disso, possui suporte nativo a áudio vídeo e geo-localização, entre outros.

Tendo em vista que tecnologias comumente utilizadas no desenvolvimento de aplicações Java para a *Web*, como o JSF (Java *Server Faces*) e JSP (Java *Server Pages*) não possuem nenhum suporte nativo a essa nova especificação, foi decidido utilizar o *framework* GWT (*Google Web Toolkit*), que, em sua versão 2.3, já possui suporte a algumas das mais novas tecnologias do *HTML5*, mesmo sendo as mais básicas.

Segundo o GOOGLE CODE *(200-a)*, com o GWT, “você cria o *front end* AJAX *(Asynchronous* Javascript *and* XML*)* na linguagem de programação Java e o GWT, então, faz a compilação cruzada para o *JavaScript* otimizado que funciona automaticamente com todos os principais navegadores”. Isso torna a tarefa de criar interfaces ricas para aplicações *Web* muito mais simples, pois resolve vários problemas de compatibilidade, que teriam de ser implementadas manualmente caso fossem utilizadas tecnologias como o *JSF*, por exemplo.

1.1 OBJETIVO GERAL

Desenvolver uma aplicação *Web* utilizando o *framework* GWT, como estudo experimental, para o acompanhamento básico de um projeto de desenvolvimento de *software* que utilize a metodologia ágil *Scrum*.

1.2 OBJETIVOS ESPECÍFICOS

Como objetivos específicos deste projeto, apresentam-se os seguintes itens:

- Desenvolver um referencial teórico sobre *Scrum*, linguagem de programação orientada a objetos para Web, *frameworks* e GWT;
- Analisar, projetar e desenvolver um protótipo para o sistema de acompanhamento, como estudo experimental das linguagens avaliadas no referencial teórico;
- Testar a aplicação e apresentar os resultados

1.3 JUSTIFICATIVA

Sempre que se inicia um novo projeto, seja ele de médio ou grande porte, uma grande fatia de tempo e esforço são aplicadas ao seu planejamento, visando diminuir os riscos de falhas futuras no projeto. Nesse tipo de caso, é muito comum que as equipes de desenvolvimento escolham o *Scrum* como metodologia para o desenvolvimento do projeto.

Segundo a SCRUM ALLIANCE (200-), o *Scrum* é um *framework* ágil para completar projetos complexos. O *Scrum* foi originalmente formalizado para projetos de desenvolvimento de *softwares*, mas funcional bem para qualquer escopo complexo e inovador de trabalho.

Mesmo assim, o *Scrum* gera uma quantidade considerável de documentação, como o *Product Backlog* e o *Sprint Backlog*, uma ferramenta para controle de todo o processo definido seria bem-vinda, facilitando ainda mais a aplicação desse *framework* e economizando ainda mais tempo.

Tendo em vista que são poucas as ferramentas que auxiliam no controle do *Scrum*, e que, as poucas disponíveis são pagas, inclusive algumas vezes possuem valores muito altos, este trabalho vem com 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.

2 REVISÃO BIBLIOGRÁFICA

Este capítulo apresenta uma revisão bibliográfica desenvolvida sobre as tecnologias, linguagens e paradigmas de gestão de projetos e programação utilizados para a construção do estudo experimental. Utilizou-se, como base para a pesquisa, livros, artigos e sites da Internet que apresentassem informações confiáveis de renomados pesquisadores da área.

2.1 SCRUM

Segundo SCHWABER (1995), o *Scrum* é um *framework*[1] para desenvolver e manter produtos complexos. O *Scrum* é formado por equipes de indivíduos e seus papéis, artefatos e regras associadas. Cada componente do *Scrum* serve para um propósito especifico 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 que se deve 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 (SCHWABER, 1995).

Três pilares sustentam a implementação empírica do controle de processo. São eles: 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 (SCHWABER, 1995).

A inspeção diz que os usuários do *Scrum* devem frequentemente inspecionar os artefatos do *Scrum* e o seu progresso, com objetivo de detectar possíveis variações indesejadas (SCHWABER, 1995).

A adaptação diz que se o inspetor determinar que o produto não esteja aceitável, o processo e o material que esta sendo processado deve ser ajustado.

[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)

Este ajuste deve ser feito o quanto antes, para evitar desvios futuros (SCHWABER, 1995).

Segundo SCHWABER (1995), a Equipe de *Scrum* é formada pelo *Product Owner*, Equipe de Desenvolvimento e *Scrum Master*.

As equipes são auto-organizadas, e sendo assim, escolhem a melhor forma de realizar o seu trabalho, ao invés de serem dirigidas por outros de fora da equipe, que, na maioria das vezes não conhecem a realidade da própria equipe. A própria equipe tem todas as qualificações necessárias para não precisar ser dirigida por nenhuma pessoa externa a equipe. As equipes do *Scrum* produzem iterativamente e de forma incremental, aproveitando ao máximo as oportunidades para realimentação. Entregas incrementais de produtos “Prontos” garantem que sempre uma versão potencialmente útil do produto ou de parte dele seja entregue ao cliente final (SCHWABER, 1995).

Segundo SCHWABER (1995), o *Product Owner* é responsável por maximizar o valor do produto e do trabalho da equipe, mas a metodologia utilizada para isto varia amplamente entre organizações, equipes e indivíduos. O *Product Owner* é responsável pelo gerenciamento do *backlog* do produto, que inclui:

- 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 pela equipe de desenvolvimento;
- 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. Ele pode representar os desejos ou objetivos de um comitê no *backlog* do produto, mas somente o *Product Owner* é responsável por mudar algo no *backlog* *(*SCHWABER, 1995*).*

Stakeholders são todos os indivíduos envolvidos no processo, como o *Product Owner*, a equipe de desenvolvimento e até mesmo o cliente (SANTOS, 200-).

A equipe de desenvolvimento 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 da equipe de desenvolvimento 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 consequentemente, podem ter dificuldades em entregar um produto potencialmente incremental (SCHWABER, 1995).

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, práticas e regras do *Scrum* (SCHWABER, 1995).

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. Eles são uma oportunidade para inspecionar e adaptar algo, e foram feitos justamente para manter e permitir a transparência crítica e inspeção. Exemplos de eventos do Scrum são a Reunião de Planejamento e a Reunião Diária (SANTOS, 200-).

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*. Resumidamente, um Sprint é um conjunto de requisitos a serem desenvolvidos pela equipe em determinado tempo. O *Sprint* é definido pelo *Scrum Master*, que também é responsável por remover qualquer coisa que impeça a equipe de entregar o *Sprint* (SANTOS, 200-).

2.2 JAVA

A *Sun Microsystems* começou o desenvolvimento da linguagem Java em meados da década de 90, originalmente sob o nome de Oak, com o lançamento da primeira versão em 1995 (LINDHOLM, 2008). Oak foi desenvolvido pela equipe de James Gosling como uma linguagem multi-plataforma que visava criar um canal de comunicação entre dispositivos de entretenimento e multimídia da época.

Como havia pouca demanda para esse tipo de serviço naquele tempo o foco da linguagem acabou se tornando outro: a *World Wide Web*, popularmente conhecida hoje como Internet.

Dessa nova tendência nasceu um navegador chamado *WebRunner*, que mais tarde ficou conhecido como *HotJava* e a linguagem passou então a se chamar Java. A partir dessa iniciativa a Java passou a atrair cada vez mais a atenção de desenvolvedores, empresas, instituições públicas e privadas e se tornou o ponto central da plataforma da Sun Microsystems que hoje conta com uma série de produtos e especificações construídas através desta linguagem. A Figura 1 mostra como a plataforma (JSE – Java *Standard Edition*) está organizada e quais os produtos e APIs a compõe.

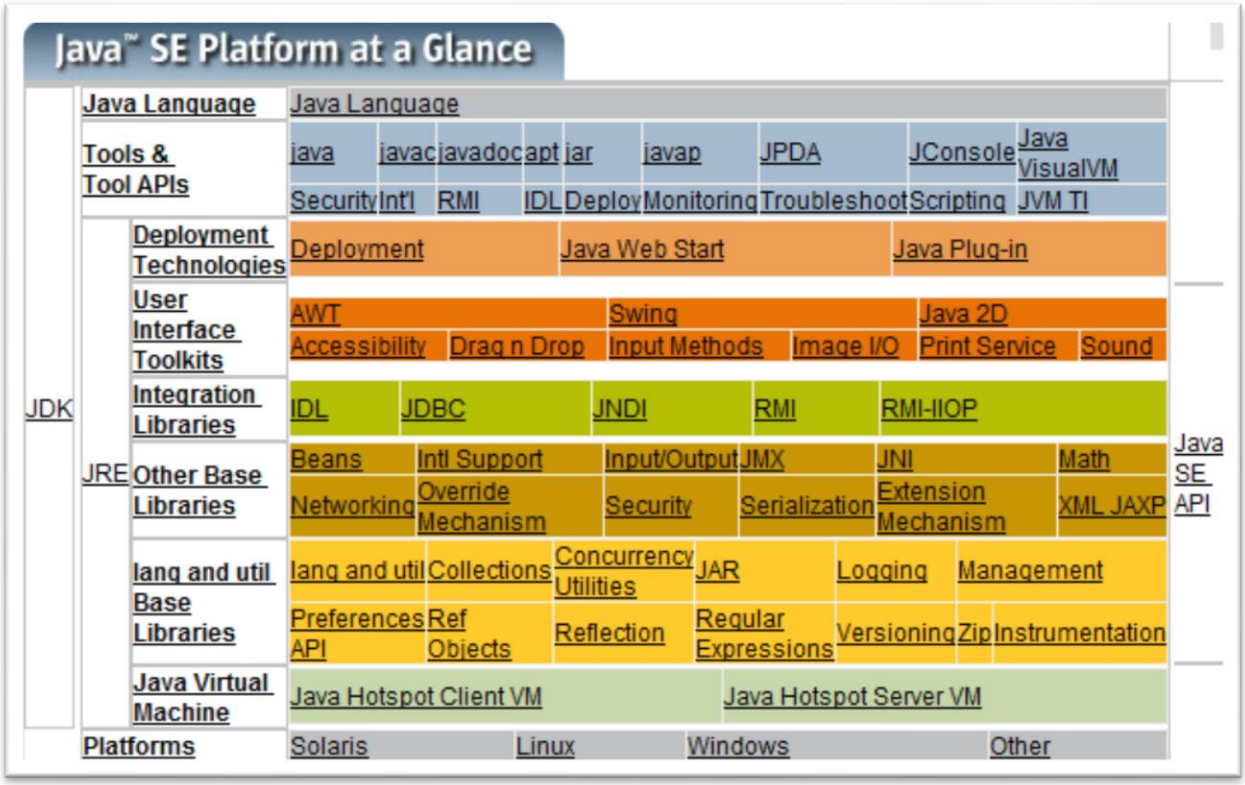


Figura 1 - Visão Geral da Plataforma Java.

Fonte: ORACLE (200-)

Dois produtos desta família merecem atenção especial e são eles o JRE (Java *Runtime Environment*) e o JDK (Java *Development Kit*). O JRE disponibiliza todos os recursos necessários (bibliotecas, Java *Virtual Machine* e *plugins* para navegadores que os capacitam a executar qualquer conteúdo escrito em Java) para executar aplicações escritas em linguagem Java. *O Software Development Kit* (SDK) tem tudo que o JRE tem, a diferença é que neste produto estão incluídas as ferramentas (compiladores e *debuggers*) necessárias para o desenvolvimento de aplicações escritas em Java. A descrição completa da plataforma (versão seis) está descrita na especificação JSR (Java *Specification Request*) 270 (JAVA COMMUNITY PROCESS, 2005).

Em 1997 a *Sun Microsystems* decidiu procurar a ISO (*International Organization for Standardization*) e logo em seguida a *Ecma International (European Computer Manufacturers Association*) para normatizar e formalizar a linguagem, mas

devido a conflitos de interesse encontradas ao longo do processo a Sun retirou seu pedido (EGYEDI, 2001).

2.3 GOOGLE *WEB* TOOLKIT

Em maio de 2006, a *Google* liberou o *Google Web Toolkit*, um conjunto de ferramentas de desenvolvimento, utilidades de programação e componentes que permitem que você crie aplicações ricas para a Internet diferente de como você fazia até então (HANSON, 2007). Após isso, a *Google* lança aproximadamente duas versões por ano, tendo lançado a versão 2.4.0 durante o desenvolvimento do presente trabalho, em oito de setembro de 2011. As principais novidades dessa nova versão, segundo o GOOGLE CODE (2011), são o fato de possui suporte ao *Google Apps Marketplace*, melhorias no GWT *Designer* e melhor suporte a dispositivos *Android*.

Ao longo dos anos, as versões do GWT estão tornando-se cada vez mais completas, com componentes melhores, mais e melhores ferramentas, correções de bugs, maior compatibilidade com diversos navegadores e cada vez mais suporte aos novos recursos do HTML*5*. Com isso, ele vem tornando-se um *framework* cada vez mais moderno e atual para o desenvolvimento *Web* sobre a plataforma Java.

2.3.1 ESTRUTURA E CONCEITOS BÁSICOS DE UMA APLICAÇÃO GWT

Uma aplicação GWT recém-criada contém uma estrutura relativamente simples, porém, bem definida, conforme é mostrado na Figura 2.

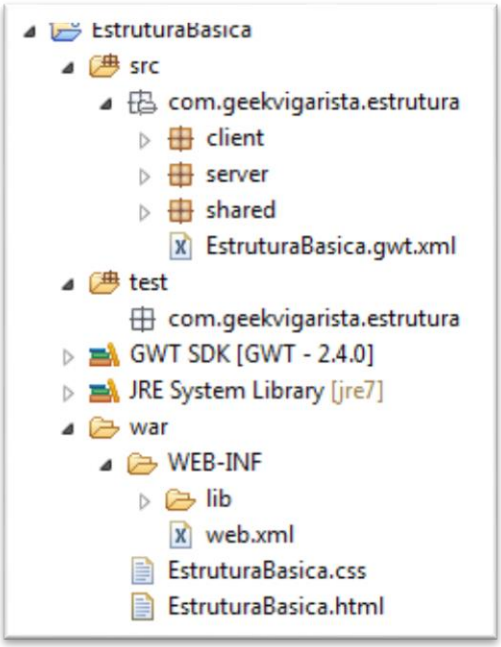


Figura 2 - Estrutura Básica de um Projeto GWT

Como pode ser observado na Figura 2, dentro do pacote principal da aplicação, existem três pacotes: *client, server, shared*. Essa organização é necessária porque o GWT converte o código da camada cliente de Java para JavaScript, assim, o GWT tentará converter tudo o que estiver nos pacotes *client* e *shared*. Os recursos contidos no pacote *shared* podem ser utilizados tanto no lado cliente quanto no lado servidor. Já os recursos do pacote *client* podem ser utilizados pelos recursos da do pacote *shared* e por recursos de sua própria camada, bem como o pacote *server* conversa com os recursos dele mesmo e com os recursos do pacote *shared*. A convenção para tudo isso é manter as classes dos objetos básicos a serem utilizados em ambas as camadas no pacote *shared*, as telas e recursos relacionados a ela no pacote *client*, e as lógicas de negócio e persistência no pacote *server*.

2.3.2 CONVERSÃO DE JAVA PARA JAVASCRIPT

O compilador do GWT compila Java para *JavaScript*, mas não o faz da mesma forma como o `javac`[2] faz. O compilador do GWT é, na verdade, um tradutor de código Java para código Javascript (COOPER, 2008). Por isso, nem tudo pode ser convertido para *JavaScript*. Apesar de o GWT ser capaz de converter todos os objetos básicos e tipos primitivos do Java para tipos semelhantes em *JavaScript*, alguns objetos compostos simplesmente não podem ser convertidos.

Para um objeto composto poder ser convertido, é necessário que sua classe, basicamente, implemente a interface `Serializable`, e tenha um construtor público sem argumentos. Seguir essas regras, porém, não significa que toda classe poderá ser convertida. Classes que utilizam a *Reflections* API do Java, ou que utilizem anotações que sejam tratadas através de *Reflections*, por exemplo, não podem ser convertidas pelo GWT. Por isso, quase sempre não é possível utilizar vários *frameworks* e APIs nativamente no GWT, pois, muitos deles (principalmente os *frameworks* de persistência) utilizam largamente anotações que são tratadas por *reflections* em classes que, teoricamente, deveriam ficar no pacote *shared* (como por exemplo, os POJOS (*Plain Old Java Objects*)), e o GWT não consegue converte-las nativamente.

Alguns *frameworks* já deram a devida importância ao GWT e criaram módulos que podem ser adicionados a aplicação para que essa conversão funcione, porém, vários outros ou simplesmente não possuem nada, ou então possuem versões ainda não funcionais em desenvolvimento. Para *frameworks* de persistência, temos como alternativa programar uma classe sem as anotações no pacote *shared*, e uma “cópia” dela, porém, com as anotações no pacote *server*, assim, quando o objeto for persistido, pode-se converte-lo para a versão com as anotações e usá-lo à vontade no lado servidor da aplicação.

Ainda observando a Figura 2 pode-se perceber o arquivo `EstruturaBasica.gwt.xml`. Este arquivo é responsável por definir todos os módulos utilizados, a classe principal de entrada da aplicação (*EntryPoint*), o tema a ser utilizado, outros pacotes de código fonte e também configurações de alguns

[2] Compilador que converte código Java para bytecode.

módulos em especial, o que não é algo obrigatório. Este arquivo costuma ser muito simples, conforme pode ser visto na Figura 3.

```
<?xml version="1.0" encoding="UTF-8"?>
<module rename-to='estruturabasica'>
<inherits
    name='com.google.gwt.user.User' />
<inherits
    name='com.google.gwt.user.theme.clean.Clean' />
<entry-point
    class='com.geekvigarista.estrutura.client.EstruturaBasica' />
<source path='client' />
<source path='shared' />
</module>
```

Figura 3 – Código XML do arquivo de configuração EstruturaBasica .gwt.xml.

Tem-se também a pasta `war`, que contém a pasta `WEB-INF`, um arquivo CSS e um arquivo HTML. Na pasta `WEB-INF`, tem-se, além da pasta `lib` com as bibliotecas utilizadas pela aplicação, o arquivo `web.xml`. Este arquivo é responsável pelo mapeamento dos *servlets* e da página de entrada da aplicação. Um exemplo de arquivo `web.xml` pode ser visto na Figura 4.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
              http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    version="2.5" xmlns="http://java.sun.com/xml/ns/javaee">

    <servlet>
        <servlet-name>greetServlet</servlet-name>
        <servlet-class>
            com.geekvigarista.estrutura.server.GreetingServiceImpl
        </servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>greetServlet</servlet-name>
        <url-pattern>/estruturabasica/greet</url-pattern>
    </servlet-mapping>

    <welcome-file-list>
        <welcome-file>EstruturaBasica.html</welcome-file>
    </welcome-file-list>

</web-app>
```

Figura 4 – Código do arquivo web.xml de exemplo.

Também pode ser notada a presença de um arquivo HTML, que é a página inicial da aplicação. Este arquivo costuma ter um conteúdo bem básico, como pode ser visto na Figura 5. Este arquivo, basicamente, conterá, além do HTML básico da aplicação (titulo), as inclusões dos arquivos de CSS e do *JavaScript* compilado pelo GWT. Além disso, pode-se observar que dentro da *tag* `body`, existe um `iframe`, que é usado pelo GWT para gerenciar o histórico, porém, não é obrigatório.

```
<!doctype html>
<html>
  <head>
    <meta http-equiv="content-type" content="text/html; charset=UTF-8">
    <link type="text/css" rel="stylesheet" href="EstruturaBasica.css">
    <title>Web Application Starter Project</title>
    <script type="text/javascript" language="javascript"
        src="estruturabasica/estruturabasica.nocache.js"></script>
  </head>
  <body>
    <iframe src="javascript:''" id="__gwt_historyFrame" tabIndex='-1'
        style="position:absolute;width:0;height:0;border:0"></iframe>
    <noscript>
      <div >
        Your web browser must have JavaScript enabled
        in order for this application to display correctly.
      </div>
    </noscript>

    <div id="carregando">Carregando... aguarde.</div>

    </table>
  </body>
</html>
```

Figura 5 – Código do arquivo HTML contido na pasta war.

Ainda dentro do pacote *client*, há o *EntryPoint* da aplicação. O *EntryPoint* nada mais é que a classe de entrada da aplicação. Quando a aplicação é acessada, ela é a primeira classe a ser carregada e executada. Basicamente, é uma classe que geralmente possui o mesmo nome do módulo da aplicação, que deve ser configurada no arquivo `.gwt.xml`, e implementa a interface *EntryPoint*, tendo somente o método `onModuleLoad`. Nesse método colocamos tudo o que deve ser carregado antes do resto da aplicação, como, por exemplo, um *framework* de injeção de dependências, ou ainda, esconder uma tela de carregando, entre outros. Um exemplo dessa classe pode ser visto na Figura 6.

```
public class Gwt_ScrumManager implements EntryPoint
{
    private final ClientGinjector ginjector = GWT.create(ClientGinjector.class);

    @Override
    public void onModuleLoad()
    {
        DelayedBindRegistry.bind(ginjector);
        ginjector.getPlaceManager().revealCurrentPlace();
        RootPanel.get("carregando").setVisible(false);
    }
}
```

Figura 6 - EntryPoint de uma aplicação

Na figura 6 é possível observar que é obtida uma instância do *framework* de injeção de dependência para o lado cliente (*Google GIN*) através da *factory* padrão do GWT, e, logo após, essa instância é usada para exibir a tela relativa à URL (*Uniform Resource Locator*) atual, através do método `revealCurrentPlace`. Por fim, a `div` carregando é escondida, informando ao usuário que a tela inicial da aplicação está pronta.

2.3.3 CONCEITOS E RECURSOS AVANÇADOS

Como a linguagem Java é orientada a objetos, é possível, facilmente, estender os componentes padrões do GWT, bem como criar novos componentes. Também é possível criar novos eventos que serão tratados pelos componentes criados.

Para exemplificar a criação de componentes e alguns outros recursos do GWT, serão mostrados os passos para a criação de um componente de busca personalizado, que deverá ficar semelhante à Figura 7.

Figura 7 – Componente de busca terminado e sendo exibido na aplicação.

Basicamente, este componente estende o componente `HorizontalPanel` (painel que pode ser utilizado para exibir vários outros componentes na horizontal). Dentro dele, teremos mais dois outros componentes: o `TextBox` (campo de entrada de texto simples) e o `PushButton` (botão). Como é visto na Figura 7 tem-se como resultado um componente estilizado, com um texto de fundo (chamado de *placeholder*), semelhante ao que pode ser encontrado no HTML5.

A estrutura do pacote do componente deverá ficar conforme é mostrado na Figura 8.

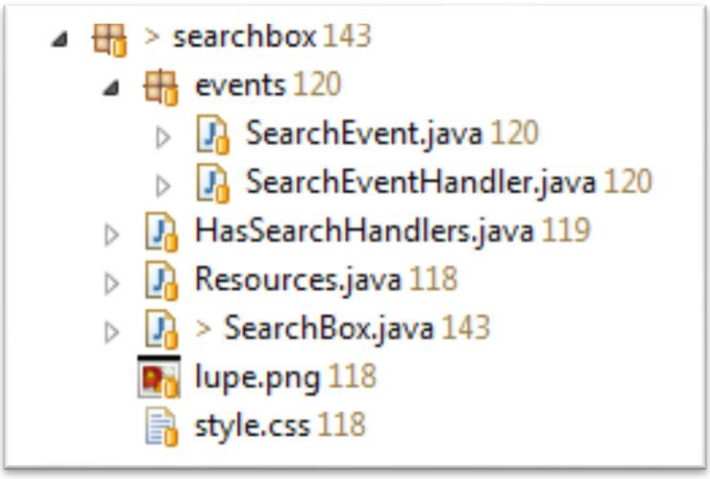


Figura 8 - Estrutura dos pacotes, classes e interfaces do componente.

Na Figura 8, pode ser notada a existência de um pacote com os eventos do componente, uma interface (`HasSearchHandlers`), que é implementada pelo componente (e que pode ser implementada também outros componentes que podem querer reaproveitar essa estrutura básica), e também, uma classe de recursos (*Resources*), que controla os recursos utilizados pelo componente (nesse caso, o ícone da lupa e o arquivo de CSS).

2.3.3.1 Criação de eventos personalizados

O GWT permite e incentiva a criação de eventos para manter o código da aplicação mais organizado. No GWT, todo evento precisa ter, no mínimo, uma classe e uma interface. A interface é relativamente simples e possui apenas os métodos que devem ser implementados pela classe que receberá o tratamento do

evento. Essa interface deve estender `EventHandler`, conforme pode ser visto na Figura 9.

```
public interface SearchEventHandler extends EventHandler
{
    void onSearch(SearchEvent event);
}
```

Figura 9 - Código da interface SearchEventHandler.

A classe, porém, é um pouco mais extensa e complicada. Ela deve estender a classe abstrata `GwtEvent`, que espera um diamante com um tipo genérico `H` estendendo `EventHandler` (a interface da figura 9). O evento também precisa ter um tipo, um método chamado `fire`, que recebe como parâmetro o gerenciador de eventos da aplicação (`EventBus`) e o método `dispatch`, que executa o método da interface `SearchEventHandler`. O código dessa classe pode ser visto na Figura 10.

```
public class SearchEvent extends GwtEvent<SearchEventHandler>
{
    private static final Type<SearchEventHandler> TYPE = new Type<Sear

    public static Type<SearchEventHandler> getType()
    {
        return TYPE;
    }

    public static void fire(EventBus eventBus)
    {
        eventBus.fireEvent(new SearchEvent());
    }

    public SearchEvent(){}

    @Override
    protected void dispatch(SearchEventHandler handler)
    {
        handler.onSearch(this);
    }

    @Override
    public Type<SearchEventHandler> getAssociatedType()
    {
        return getType();
    }
}
```

Figura 10 - Classe SearchEvent.

É possível notar também a existência da interface `HasSearchHandlers`. Essa interface será implementada por todas as classes que tratarão o evento `SearchEvent`, e tem como intuito principal a melhor legibilidade e qualidade do código, não sendo sua criação obrigatória na construção de um componente qualquer. Como pode ser visto na Figura 11, ela é bastante simples, apenas estendendo `HasHandler` e possuindo um método que retorna um objeto do tipo `HandlerRegistration`.

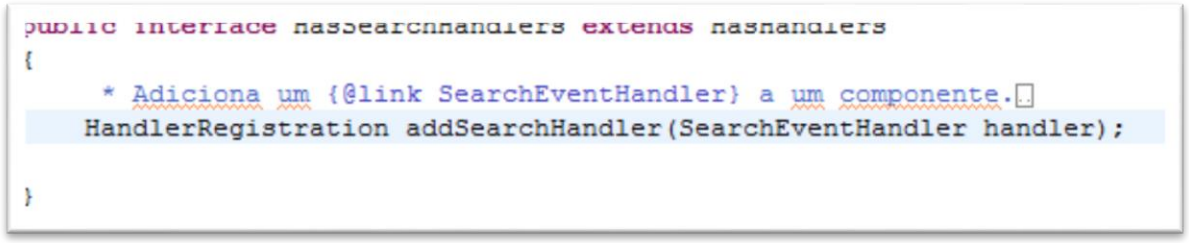


Figura 11 - Interface HasSearchHandlers

2.3.3.2 Gerenciando recursos com o ClientBundle

O GWT possui também outro recurso muito interessante, chamado *ClientBundle*. Ele permite que o desenvolvedor “ligue” arquivos e classes CSS ou imagens em uma interface. Depois, pode ser obtida uma instância dessa interface através da *factory* padrão do GWT. Com essa instância, o desenvolvedor pode facilmente injetar estilos e imagens na aplicação. Estes estilos e imagens são automaticamente removidos do HTML da aplicação assim que o GWT decidir que eles não são mais necessários ou não estão mais sendo utilizados, o que ajuda a manter o HTML mais simples, menor e o mais leve quanto possível.

Dentro do `ClientBundle`, para tratar os estilos, a *Google* recomenda o uso do `CssResource`. Para isso, simplesmente cria-se uma interface que estenda `CssResource` dentro ou não da interface que estende `ClientBundle`, e coloca-se dentro dela métodos públicos retornando *String* com os nomes das classes CSS que se deseja utilizar. Depois, na interface que estende `ClientBundle`, cria-se um método que retorna uma instância da interface que estende `CssResource`, que deve ser anotado com `@Source`, passando o nome do arquivo CSS a ser utilizado.

Juntamente com os recursos para CSS, existe o `ImageResource`. Ele tem o mesmo uso do `CssResource`, porém, é usado para imagens.

Um exemplo de código da interface `Resources` pode ser visto na Figura 12.

```
 * Interface do campo de busca.
public interface Resources extends ClientBundle
{

    Resources INSTANCE = GWT.create(Resources.class);
    SearchCss CSS = INSTANCE.style();

     * interface com os estilos do css.
    public interface SearchCss extends CssResource
    {
        String box();

        String focus();

        String blur();

        String button();
    }

    @Source("style.css")
    SearchCss style();

    @Source("lupe.png")
    ImageResource lupe();
}
```

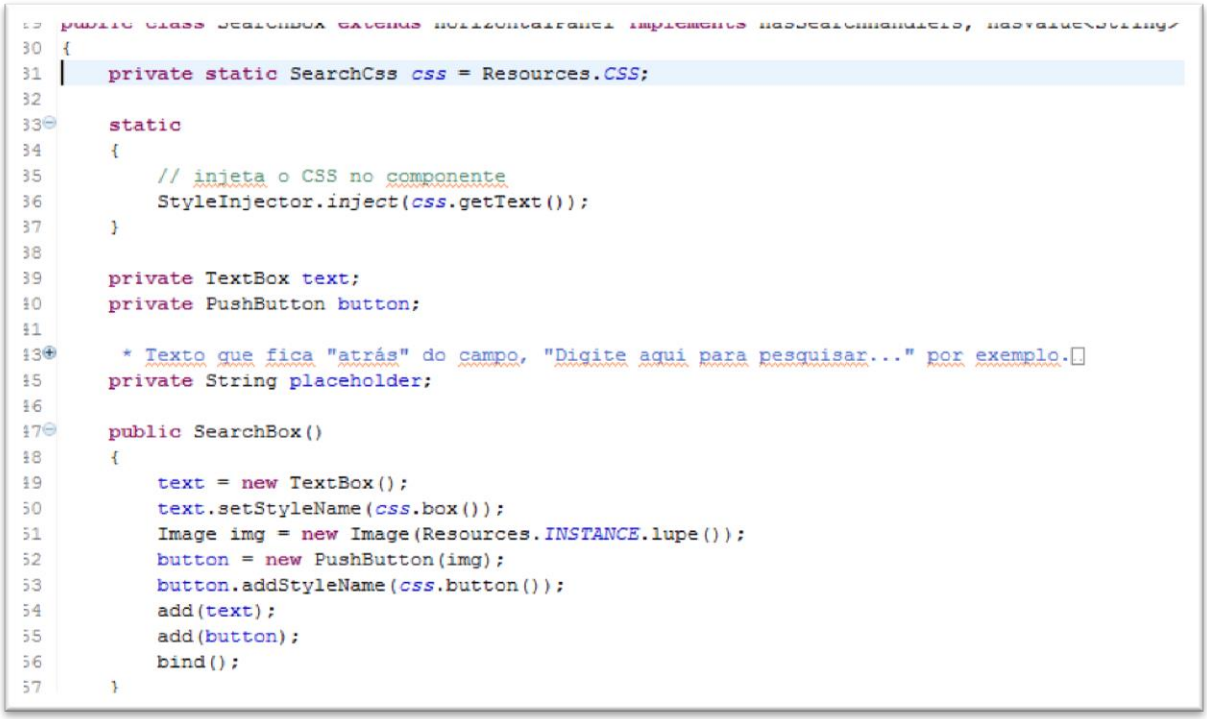
Figura 12 - Interface Resources, estendendo ClientBundle.

Conforme pode ser visto na Figura 12, pode-se notar facilmente que a interface `SearchCss` irá carregar o arquivo `style.css`. Este arquivo deverá conter todas as classes com os nomes idênticos aos nomes dos métodos da interface `SearchCss`. Isso é necessário, porque o GWT “amarra” a interface com o arquivo CSS, tornando assim possível carregar as classes CSS a partir dos métodos dessa interface. O uso do `CssResource` também permite que o desenvolvedor tenha mais alguns recursos nos arquivos CSS, como constantes, condições e referências a URLs, entre outros que não convém exemplificar aqui. O código do arquivo CSS do componente em questão pode ser visto na Figura 13.

```
@def boxHeight 16px;
@def boxRadius 5px;
.box {
    \-moz-border-radius: boxRadius;
    \-webkit-border-radius: boxRadius;
    padding: 5px;
    width: 100%;
    height: boxHeight;
    border: 1px solid #A9ADB8;
}

.button {
    margin-left: 6px;
    width: 17px;
}

.focus {
    color: #000;
    font-style: normal;
}

.blur {
    color: #777;
    font-style: italic;
}
```

Figura 13 - Arquivo style.css do componente de busca.

Após tudo isso, a classe do componente em si acaba ficando relativamente simples, com menos de 200 linhas. O código básico da classe pode ser visto na Figura 14.



```
30  {
31      private static SearchCss css = Resources.CSS;
32
33      static
34      {
35          // injeta o CSS no componente
36          StyleInjector.inject(css.getText());
37      }
38
39      private TextBox text;
40      private PushButton button;
41
43       * Texto que fica "atrás" do campo, "Digite aqui para pesquisar..." por exemplo.
45      private String placeholder;
46
47      public SearchBox()
48      {
49          text = new TextBox();
50          text.setStyleName(css.box());
51          Image img = new Image(Resources.INSTANCE.lupe());
52          button = new PushButton(img);
53          button.addStyleName(css.button());
54          add(text);
55          add(button);
56          bind();
57      }
```

Figura 14 - Parte do código da classe SearchBox

Conforme pode ser observado na Figura 14, na linha 36 é invocado o método estático `inject` da classe `StyleInjector`. Esse método garante que o conteúdo do arquivo CSS será injetado na página de forma segura. Também pode-se observar o construtor da classe montando as partes do componente de forma muito semelhante às tecnologias para desenvolvimento *desktop* com Java, como o *Swing*, por exemplo. É possível observar também o uso do `ImageResource` na linha 51.

Ainda observando a Figura 13, pode ser visto que a classe implementa as interfaces `HasSearchHandlers` e `HasValue<String>`. Na Figura 15 pode-se observar a implementação dos métodos dessas interfaces.

```
45      @Override
46      public HandlerRegistration addSearchHandler(final SearchEventHandler handler)
47      {
48          return addHandler(handler, SearchEvent.getType());
49      }
50
51      @Override
52      public HandlerRegistration addValueChangeHandler(ValueChangeHandler<String> handler)
53      {
54          return text.addValueChangeHandler(handler);
55      }
56
57      @Override
58      public String getValue()
59      {
60          return text.getValue();
61      }
62
63      @Override
64      public void setValue(String value)
65      {
66          text.setValue(value);
67          if(!value.equals(placeholder))
68          {
69              setFocusStyle();
70          }
71      }
72
73      @Override
74      public void setValue(String value, boolean fireEvents)
75      {
76          text.setValue(value, fireEvents);
77      }
```

Figura 15 - Implementação dos métodos das interfaces HasSearchHandlers e HasValue<String>.

O restante do código da classe trata da lógica para mostrar ou não o *placeholder*, não convém mostrá-lo aqui pois não é relevante ao assunto tratado.

2.3.3.3 Internacionalização

Desenvolver aplicações que podem ser usados em diferentes países e idiomas requer a aplicação de técnicas específicas, mas o GWT simplifica lidar com as questões de internacionalização (i18n[3]) e localização (l10n[4]) (KEREKI, 2010).

Para tornar uma aplicação GWT internacionalizável, é necessário uma interface de constantes que representarão as expressões e/ou palavras a serem internacionalizadas. Faz-se necessária também a criação de arquivos de propriedades para cada idioma suas respectivas “versões” das expressões.

[3] A expressão i18n é uma espécie de acrônimo para “*internationalization*” (internacionalização), o número “18” significa que existem 18 letras entre o primeiro “i” e o último “n” da palavra.
[4] De forma semelhante ao i18n, l10n significa “*localization*” (localização), e o número “10” significa que existem 10 letras entre o primeiro “l” e o último “n” da palavra.

No Quadro 1 é possível observar as propriedades para três idiomas. Nota-se que o primeiro arquivo (`Usuario.properties`), que não contém nenhuma especificação em seu nome quanto à qual idioma ou localização ele está relacionado, é o idioma padrão da aplicação. Já o segundo arquivo, chamado de `Usuario_en_US.properties`, especifica tanto o idioma (*en*) quanto a localização (US – *United States*). Se fosse necessário fazer outra tradução para o inglês britânico, por exemplo, poderia-se criar outro arquivo, chamado `Usuario_en_GB.properties` e fazer as traduções necessárias. O terceiro arquivo contém uma possível tradução para o Alemão (*de*).

Usuario.properties	Usuario_en_US.properties	Usuario_de.properties
user=Usuário	user=Username	user=Benutzername
passwd=Senha	passwd=Password	passwd=Passwort
Name=Nome	Name=Name	name=Name

Quadro 1 - Exemplo de propriedades para três idiomas diferentes.

Também deve ser criada a interface necessária para fazer a ponte entre os arquivos de propriedades e as expressões a serem utilizadas no código (Figura 16). Essa classe deve, basicamente, estender `Constants`, e conter métodos públicos retornando *String* com os nomes das expressões que se deseja utilizar no restante da aplicação.

```
public interface Mensagem extends Constants
{
    public String salvar();

    public String concluir();

    public String cancelar();

    public String nome();
```

Figura 16- Trecho do código de uma interface de intercionalização.

Após isso, para obter-se uma propriedade da interface, pode-se utilizar a *factory* padrão do GWT, como pode ser visto na Figura 17.

```
Mensagem instance = GWT.create(Mensagem.class);
System.out.println(instance.nome());
```

Figura 17 - Exemplo de como obter a instância da interface de internacionalização e de como obter uma propriedade dela.

Dessa forma, todas as mensagens da aplicação ficam centralizadas, e podem ser acessadas da mesma forma, não fazendo diferença qual idioma ou qual local o usuário fala ou está, bastando apenas o idioma e/ou a localização estarem implementados. No caso do idioma não ser encontrado, o GWT automaticamente usa o idioma padrão da aplicação.

2.4 MODEL-VIEW-PRESENTER X MODEL-VIEW-CONTROLLER

Esta seção mostrará as principais diferenças entre dois padrões estruturais, o MVC e o MVP.

2.4.1 MODEL-VIEW-CONTROLLER

Desde o *Smalltalk*[5] na década de 80, o padrão MVC (*Model-View-Controller*) tem sido usado em muitos casos, principalmente para projetar interfaces com o usuário (KEREKI, 2010). Um sistema utilizando MVC, segundo KEREKI (2010), é composto basicamente por:

- **Model**: ***Model***: Compreende toda a lógica de negócios. Para sistemas baseados na *Web*, isso significa *servlets*, *Web services* e afins;
- ***View***: Possui todos os componentes necessários para a interação com o usuário;

[5] Linguagem orientada a objetos fracamente tipada. Nela tudo é objeto e não existem tipos primitivos.

- ***Controller***: Fica entre a *View* e a *Presenter* e traduz as ações do usuário na *View* para o atualizações na *Model*. Dependendo do *framework* utilizado, a camada *Controller* poder ficar tanto no lado servidor quanto no lado cliente.

O modo como esses componentes se relacionam entre si pode ser observado na Figura 18.

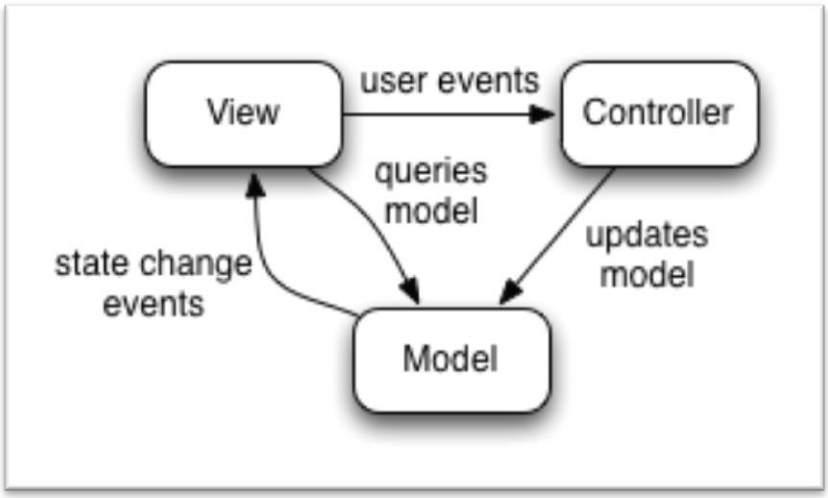


Figura 18 – Modelo MVC

Fonte: Google Code (200-e)

Observando a Figura 18, pode-se notar que, basicamente, a *Controller* observa a *View*, e em resposta a eventos do usuário, ele pode desencadear mudanças na *Model* (através do envio de comandos à *Model*) ou atualizar a *View* para mostrar os resultados de métodos ou eventos. A *View* pode enviar consultas para a *Model* (para obter dados) e também observar eventos de alteração de modelo, para, eventualmente, atualizar a si própria. Finalmente, a *Model* recebe comandos de atualização da *View*, responde a consultas a partir da *View*, e se comunica mudanças no modelo para a *View*. (KEREKI, 2010)

Ainda segundo KEREKI (2010), o desenvolvimento com GWT utilizando MVC gera diversas dificuldades na interação entre a *View* e a *Model*, pois, segundo ele, fazer com que a *Model* informe atualizações para a *View* se tornaria algo muito complicado por causa da separação cliente/servidor do GWT. O mesmo acontece

fazendo com que a *View* envie requisições para a *Model*, pois torna os testes mais difíceis.

2.4.2 MODEL-VIEW-PRESENTER

Recentemente, uma variante do MVC em que a *Controller* é substituída por uma *Presenter*, que possui mudança de responsabilidades, e provou ser mais apropriada para aplicações GWT, o MVP (*Model-View-Presenter*).

O MVP teve origem nos anos 90, na *Taligent*, um empreendimento em conjunto das empresas *Apple*, IBM e HP. O padrão foi mais tarde migrado para a linguagem Java, e popularizado através do artigo escrito pelo então CTO da *Taligen*, Mike Potel. Depois da falência da *Taglient* em 1997, Andy Bower e Blair McGlashan adaptaram o MVP para ser usado como base para o *Smalltalk User Interface Framework*. Em 2006, a *Microsoft* começou a incorporar o padrão MVP na documentação e nos exemplos do *.NET Framework*. Desde então, várias variantes foram surgindo para se adaptarem as linguagens e *frameworks* as quais se deseja aplicá-la.

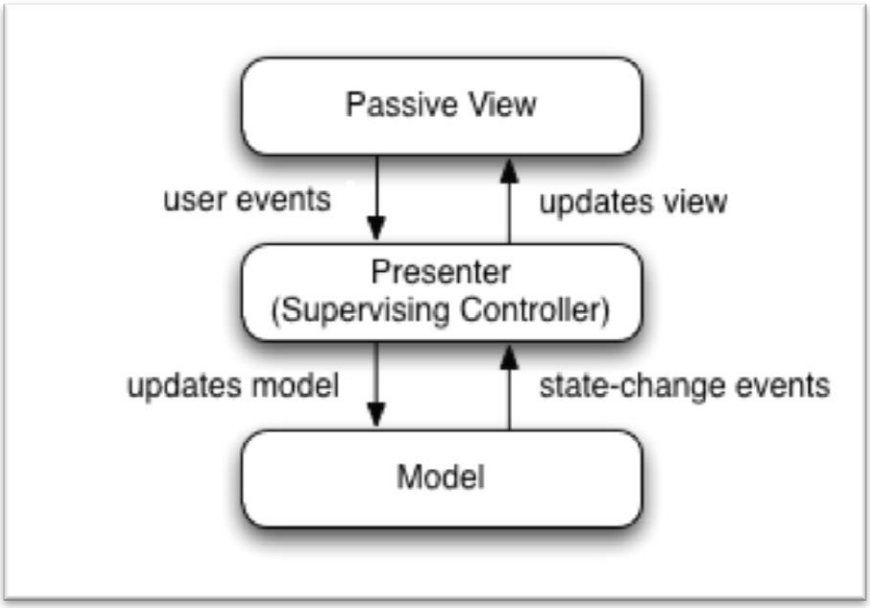


Figura 19 - Modelo MVP

Fonte: Google Code (200-e)

A forma como as camadas se comunicam no MVP pode ser observada na Figura 19. Segundo KEREKI (2010), no MVP, as responsabilidades de cada camada diferem um pouco do MVC:

- ***Model***: Possui a mesma responsabilidade que no MVC, porém ele se comunica apenas com a *Presenter*;
- ***View***: Possui uma responsabilidade semelhante à do MVC, mas não se comunica com a *Model*. Sempre que o usuário efetua alguma ação, a *View* informa a *Presenter* sobre o evento, que pode pedir à *View* para se atualizar;
- ***Presenter***: É fundamental para esse padrão, pois é a ponte entre a *Model* e a *View*. Em resposta aos eventos do usuário, ele pode se comunicar com a *Model*, e dependendo de sua resposta, enviar comandos para atualizar a *View*.

Neste padrão, a *View* é simples e praticamente não tem lógica. Ela conterá apenas códigos para criar e exibir componentes, para obter ou definir os seus valores e despachar os eventos do usuário para a *Presenter*.

2.5 DEPENDENCY INJECTION

Segundo VANBRABANT (2008), nos últimos anos tem havido uma série de novidades em torno de IoC (*Inversion of Control*) e DI (*Dependency Injection*). Olhando o passado de toda essa terminologia, usar DI freqüentemente significa que, em vez de obter suas dependências, você pode optar por recebê-las de algum lugar, e você não se importa de onde elas vêm.

Segundo VANBRANT (2008), as pessoas muitas vezes explicam isso como o Princípio de *Hollywood* – não nos chame; nós vamos chamá-lo. Este é o conceito básico da IoC. Já a DI é uma técnica utilizada para instanciar classes concretas sem se manter acoplado a elas (VANBRABANT, 2008). De forma resumida, IoC é a estratégia, DI é a prática, embora não seja a única.

Em uma solução utilizando DI, as dependências entre os módulos são definidas através da configuração de uma infraestrutura (chamada de *container*), que é responsável por “injetar” em cada componente as dependências declaradas.

As principais vantagens da utilização de DI, segundo VANBRABANT (2008), são:

- As dependências tornam-se imediatamente visíveis ao se olhar para a estrutura da classe;
- Torna-se fácil utilizar várias implementações de um mesmo módulo ou classe dentro da mesma aplicação;
- Livrar-se da invocação de um método estático em um *factory* é sempre uma coisa boa. Chamadas em métodos estáticos são difíceis de testar, porque não é possível mudar o comportamento real, como é possível utilizando interfaces. É sempre bom livrar-se de um método estático;
- Casos de teste tornam-se mais fáceis de escrever.

E ainda, segundo PRASANNA (2009), DI elimina a necessidade dos clientes de conhecer suas dependências e como criá-las, assim como as *factories* fazem. Ele deixa para trás os problemas de estado compartilhado e desordem repetitiva, movendo a responsabilidade de construção para uma biblioteca.

Existem vários *frameworks* de DI no mercado, falaremos de um deles, o *Google Guice*, e de sua variação específica para uso com o GWT, o *Google GIN*.

2.5.1 GOOGLE GUICE

O *Google Guice* (pronuncia-se *Juice*), segundo o GOOGLE CODE (200-c), diminui a necessidade de criar *factorys* e de usar a *keyword* new em seus projetos Java. A página do projeto inclusive brinca com a expressão "`@Inject` *is the new* `new`". Apesar dessa expressão, em alguns casos, continuará sendo necessário escrever *factories* e/ou utilizar o `new`, mas o código não dependerá exclusivamente deles.

O *Guice* utiliza de forma abrangente os recursos da linguagem Java, como o *type-safe*, *generics*, *annotations* e *reflections*. Seu principal objetivo é tornar seu código fácil de testar e *debuggar*, bem como agilizar o desenvolvimento (GOOGLE CODE, 200-c). A própria *Google* utiliza o *Guice* em suas aplicações desde 2006.

É plenamente possível fazer DI manualmente. Porém, usar o *Guice*, segundo VANBRABANT (2008), possui várias vantagens, entre elas:

- Tirar proveito de seu gerenciamento de tempo de vida de objeto automatizado;
- Pelo fato de não ser necessário expressar os eventos entre objetos diretamente no código, é possível facilmente reutilizar ou substituir estes eventos em toda a aplicação e além dela;
- Torna-se possível detectar erros de dependências erradas ou faltantes mais rapidamente;
- Uma fez que o *framework* estiver controlando a vida de um objeto torna-se possível fazer todo tipo de coisas com ele, como aplicar os conceitos de AOP (*Aspect Oriented Programming*);
- Escreve-se menos código;
- Obtém-se uma estrutura cuidadosamente elaborada que irá ajudá-lo a cair no "abismo do sucesso".

Ainda segundo VANBRABANT (2008), “nós vivemos em uma época onde a escrita de *software* para um determinado conjunto de requisitos não é mais suficiente. Precisamos escrever *software* de fácil manutenção, fácil de testar e fácil de ler. Hoje em dia, gastamos muito mais tempo lendo, refatorando e reutilizando código existente do que escrevendo código novo”.

Utilizar o *Guice* é uma tarefa relativamente simples. A título de exemplo, imaginemos um sistema com diferentes implementações de um mesmo DAO (*Data Access Object*), para diferentes *frameworks* de persistência. Antes de qualquer coisa, é necessário uma interface que deverá ser implementada pelas diversas implementações desse DAO. Usando como exemplo um `DaoUsuario`, teremos então uma interface chamada `IDaoUsuario`, como pode ser observado na Figura 20.

```
public interface IDaoUsuario extends IDao<Usuario, UsuarioPOJO>
{
    List<Usuario> buscarLike(String parametro);

    List<Usuario> buscaByLogin(String login, int limite);

    Usuario login(String login, String senha);
}
```

Figura 20 - Código da interface IDaoUsuario

Assim, todas as diferentes implementações do `DaoUsuario` para diferentes *frameworks* de persistência devem implementar esta interface. Um exemplo pode ser observado na Figura 21.

```
public class DaoUsuario extends BasicDAO<UsuarioPOJO, ObjectId> implements IDaoUsuario
{

    @Inject
    public DaoUsuario()
    {
        super(MongoConnection.getDatastore());
    }

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

    /**
     * Metodo que salva um Usuario.
```

Figura 21 - Trecho de código de uma implementação da interface IDaoUsuario

Na Figura 21, é possível perceber a presença da anotação `@Inject`. Essa anotação diz ao *Guice* que o construtor anotado deve ser o construtor utilizado para se obter uma nova instância dessa classe. Segundo VANBRABANT (2009), somente um construtor de uma classe pode ser anotado com essa anotação. A intenção disso é manter a imutabilidade da classe e das dependências obrigatórias. Ainda segundo ele, é possível anotar métodos e parâmetros, porém, havendo dois ou mais *setters*, por exemplo, anotados, não é possível garantir qual deles será injetado antes ou depois.

Continuando com o exemplo, torna-se também, necessário criar os módulos que definam as classes que estão implementando cada interface. Isso pode ser feito facilmente criando uma classe que estende `AbstractModule`, como pode ser observado na Figura 22.

```
public class DAOModule extends AbstractModule
{
    @Override
    protected void configure()
    {
        bind(IDaoUsuario.class).to(DaoUsuario.class);
```

Figura 22 - Trecho de código de um módulo do Guice.

Claro que essa é a forma mais básica de se fazer isso. É possível ainda, além de outras coisas, definir o ciclo de vida de uma instância de um objeto como sendo um *Singleton*. Isso pode ser observado na Figura 23.

```
public class DAOModule extends AbstractModule
{
    @Override
    protected void configure()
    {
        bind(IDaoUsuario.class).to(DaoUsuario.class).in(Singleton.class);
```

Figura 23 - Trecho de código de um módulo do Guice, utlizando Singleton.

Após isso, injetar uma instância de qualquer classe definida no módulo torna-se muito fácil. Isso pode ser visto na figura 24.

```
Injector inj = Guice.createInjector(new DAOModule());
IDaoUsuario dao = inj.getInstance(IDaoUsuario.class);
```

Figura 24 - Obtendo uma instância de IDaoUsuario através de um módulo do Guice

Na Figura 24, caso fosse preciso utilizar outra implementação, seria necessário somente alterar o módulo a ser utilizado. O restante do código permaneceria intacto.

2.5.2 GOOGLE GIN

O *Google GIN* (GWT *INjection*) visa prover injeção de dependências para o lado cliente das aplicações GWT, e é construído sobre o *Guice*.

O motivo pelo qual o GIN foi criado é simples. Segundo o GOOGLE CODE (200-d), o modo como o *Guice* é usado não funciona no lado cliente porque o *Guice* faz muito uso de recursos da *Reflections* API do Java, e, muitos desses recursos,

não são emulados pelo GWT no lado cliente. O GIN, então, de alguma forma, contorna esse problema, fazendo com que seja possível usar DI também no lado cliente, com o custo de algumas linhas de código a mais em relação ao *Guice*. Apesar disso, a configuração do GIN, segundo o GOOGLE CODE (2008), pode ser feita seguindo cinco passos.

Primeiro, deve-se herdar o módulo do GIN. Para isso, deve-se importar o módulo do GIN no arquivo `.gwt.xml` da sua aplicação, conforme a Figura 25.

```
<inherits name="com.google.gwt.inject.Inject"/>
```

Figura 25 - Importando o Módulo do GIN.

Deve-se também definir o que o *framework* chama de `Ginjector`, que nada mais é que o "injetor do GIN". Para isso, deve-se declarar uma interface com métodos que retornem os tipos desejados. Um exemplo pode ser visto na Figura 26.

```
@GinModules({DispatchAsyncModule.class, ClientModule.class
public interface ClientGinjector extends Ginjector
{
    PlaceManager getPlaceManager();

    EventBus getEventBus();

    /*
     * Providers das presenters
     */
    Provider<MainPresenter> getMainPresenter();

    AsyncProvider<HomePresenter> getHomePresenter();
```

Figura 26 - Trecho de código da classe que estende Ginjector.

Também é necessário declarar as ligações entre as interfaces e suas implementações. Isto pode ser feito de forma semelhante ao *Guice*. A diferença

básica é que, ao invés de estender `AbstractModule`, estende-se `AbstractGinModule`, conforme pode ser observado na Figura 27.

```
public class UtilsModule extends AbstractGinModule
{

    @Override
    protected void configure()
    {
        bind(IStatusPopupPanelHandler.class)
            .to(StatusPopupPanelHandler.class)
            .in(Singleton.class);
    }

}
```

Figura 27 - Código de um módulo a ser usando pelo GIN.

Deve-se também associar os módulos da aplicação ao `Ginjector`. Isso pode ser feito através da anotação `@GinModules`, conforme pôde ser observado na Figura 26.

O último passo é a instanciação do injetor, que é feita utilizando a factory padrão do GWT, como pode ser visto na Figura 28.

```
private final ClientGinjector ginjector = GWT.create(ClientGinjector.class);
```

Figura 28 - Exemplo de instanciação do ginjector.

Após isso, torna-se simples obter a instância do objeto a ser injetado. Seguindo o exemplo da Figura 26, o código ficaria semelhante ao da Figura 29.

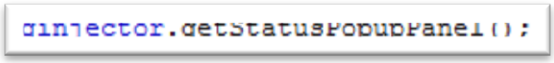


Figura 29 - Obtendo a instância do StatusPopUpPanel.

A principal vantagem de se utilizar o GIN, é que ele provê uma solução bastante simples para DI no lado cliente de uma aplicação GWT que já usa Guice. Segundo o GOOGLE CODE (2011), vários times de desenvolvimento da Google já utilizam o conjunto Guice e GIN em aplicações em produção, o que aumenta a confiabilidade do *framework*, levando em consideração que já existem diversas aplicações em produção usando-o, como o *AdWords*[6], por exemplo.

[6] https://adwords.google.com

2.6 GWT-PLATFORM

Segundo a página do projeto no GOOGLE CODE (200-b), o "*GWT-Platform* é uma estrutura MVP completa para seu próximo projeto GWT". Ele é um *fork* de dois outros *frameworks* para GWT: o *GWT-Presenter* e o *GWT-Dispatch*. O *GWT-Presenter*, é uma implementação da parte *Presenter* do modelo MVP sugerido por na sessão "*Google Web Toolkit Architecture: Best Practices For Architecting Your GWT App*" de Ray Ryan, no *Google I/O 2009* (GOOGLE CODE, 200-g). O *GWT-Dispatch* foi inspirado nessa mesma sessão, porém, implementa uma solução para a parte *Command Pattern* (GOOGLE CODE, 200-f).

O GWTP (*GWT-Platform*, pronuncia-se "*gooteepee*") possui uma solução completa para injeção de dependências, tanto no lado cliente como no lado servidor (utilizando o combo GIN e *Guice*, explicados anteriormente), um gerenciamento simples e eficiente do histórico do navegador, um *Command Pattern* reutilizável para chamadas RPC (*Remote Procedure Call)* assíncronas, e uma forma de implementação em que o desenvolvedor é forçado a utilizar o as boas práticas propostas pelo modelo MVP e por diversos engenheiros da *Google* (BEAUDOIN, 2011).

Esse conjunto de recursos previamente testados e aninhados, visam facilitar a vida do desenvolvedor, levando em consideração que a aplicação será implementada seguindo as boas práticas sugeridas pelas sessões de David Chandler e de Ray Ryan citadas anteriormente, ou seja, o desenvolvedor estará usando os melhores padrões de arquitetura e que proporcionam o melhor desempenho para a aplicação.

O GWTP possui um grande conjunto de classes abstratas, interfaces e outras soluções que visam manter um padrão sólido e correto de desenvolvimento, melhorando a manutenibilidade e legibilidade do código.

Como o GWTP é composto por diversas soluções, elas serão explicadas detalhadamente nos próximos capítulos, sempre que possível com exemplos de códigos e de utilização.

2.6.1 PRESENTER

A classe abstrata *Presenter* do GWTP, é, sem dúvida, a mais importante e uma das mais complexas que o desenvolvedor utilizará em suas aplicações GWTP. Isso se deve pelo fato que, como foi abordado nos capítulos anteriores, a *Presenter* é a camada que representa algo semelhante ao centro de controle da aplicação, e a *Presenter* do GWTP nada mais é que uma implementação dessa parte do modelo, baseada fortemente nos pontos debatidos por Ryan Ray em sua sessão sobre arquitetura de aplicações GWT na *Google I/O* 2009.

Qualquer desenvolvedor pode implementar o modelo MVP com GWT (ou qualquer outro *framework*) à sua maneira, porém, isso acarretaria em escrever várias linhas de código que podem não ser a melhor implementação para esse caso, ou podem conter falhas que comprometam o funcionamento da aplicação futuramente.

A implementação da camada *Presenter* do GWTP praticamente obriga o desenvolvedor a seguir diversas boas práticas e utilizar diversos padrões de projeto, tendo como principais objetivos uma arquitetura excelente, um código legível e um desempenho ótimo para o usuário final.

Utilizando a implementação do GWPT, toda *Presenter* necessita, obrigatoriamente, duas interfaces auxiliares. A primeira é relativamente simples, devendo apenas estender a interface *View*, e conter os métodos que deverão ser implementados pela classe *View* depois. Essa classe *View* representa a camada *View* do modelo MVP. Estes métodos, geralmente, retornam valores de campos de um formulário, ou permitem que, a partir da *Presenter*, o desenvolvedor adicione eventos a botões ou outros componentes.

A segunda interface necessária é um *Proxy*, que nas *Presenters* mais simples estende a interface *Proxy* do GWTP. Essa classe não necessita de nenhum método, apenas algumas anotações utilizadas pelo GWTP para gerar o código dela automaticamente. A interface *Proxy* é necessária para que o GWTP consiga criar um *singleton* leve antes mesmo de a Presenter ser instanciada. Esse *singleton* escuta os eventos da aplicação, esperando por algum que se aplique a *Presenter* em questão, como um `RevealContentEvent`, que precisa da instância da *Presenter*

para revelar a si mesma. Então, quando um evento deste tipo ocorre, o *Proxy* da *Presenter* automaticamente faz o necessário para ter a instância dela pronta.

A interface *Proxy* pode receber diversas anotações. As mais comuns são:

- **`@ProxyCodeSplit`**: Define que a *Presenter* será uma *Presenter* filha de uma *Presenter* superior. Isso será melhor explicado ao longo dos próximos capítulos deste trabalho;
- **`@ProxyStandard`**: Define que a *Presenter* substituirá todo o conteúdo da página, ou seja, ela é representará uma tela que usará toda a janela disponível do navegador do usuário;
- **`@UseGatekeeper`**: Define o *Gatekeeper* que será utilizado para verificar se essa *Presenter* pode ser exibida em uma ou mais situações previamente determinadas. Será melhor explicado ao longo deste trabalho;
- **`@NameToken`**: Define o *token* de histórico a ser relacionado com a *Presenter*. Para utilizar esta anotação, é necessário que a interface de *Proxy* da *Presenter* em questão estenda ProxyPlace ao invés de Proxy.

A *Presenter* então deve estender a classe abstrata `Presenter` do GWTP, que recebe um diamante esperando como parâmetros as interfaces `View` e `Proxy` relacionadas a ela.

Como as *Presenters* são injetadas pelo GIN, é necessário configurá-las em algum dos módulos da aplicação (Figura 30) e na classe `Ginjector`, conforme a Figura 31.

```
bindPresenter(HomePresenter.class, HomePresenter.HomeView.class,
        HomeView.class, HomePresenter.HomeProxy.class);
```

Figura 30 - Configurando uma Presenter em um modulo do GIN.

```
@GinModules({DispatchAsyncModule.class, ClientModule.class
public interface ClientGinjector extends Ginjector
{
    PlaceManager getPlaceManager();

    EventBus getEventBus();

    AsyncProvider<HomePresenter> getHomePresenter();

    AsyncProvider<AddUserPresenter> getAddUserPresenter();
```

Figura 31 - Trecho de código da classe Ginjector.

Feito isso, pode-se exibir a *Presenter* de três maneiras distintas. Na primeira, caso a *Presenter* seja um *Place*, acessa-se manualmente a URL relativa à *Presenter* em questão. Na segunda, usa-se um componente Hyperlink ligado ao *token* ligado a *Presenter*. Na terceira e última, cria-se um objeto do tipo `PlaceRequest`, e usa-se a instância de `PlaceManager` para exibi-la.

Segundo o GOOGLE CODE (200-b), às vezes o desenvolvedor precisará executar diversas operações em momentos chaves do ciclo de vida de uma *Presenter*. Para este fim, o GWTP fornece alguns "ganchos" em forma de métodos que podem ser sobrescritos. São eles:

- **`onBind`**: Executado logo após a construção da classe. Comumente utilizado para adicionar *handlers* de cliques ou semelhantes na implementação da *View*;
- **`onUnbind`**: Executado quando a *Presenter* é liberada pela aplicação. Desfaz qualquer operação feita no *onBind*, exceto a adição de handlers;
- **`onReveal`**: Executado quando a *View* relacionada à *Presenter* em questão é exibida ao usuário;

- **`onHide`**: Executado quando a *View* relacionada à *Presenter* em questão é oculta;
- **`onReset`**: Executado quando ocorre uma navegação e a *View* da *Presenter* em questão continua visível.

Todos estes “ganchos” fornecem ao desenvolvedor maneiras práticas e padronizadas de tratar as mais variadas situações e acontecimentos na aplicação.

Muitas vezes torna-se necessário manter várias *Presenters* aninhadas, ou seja, uma ao lado da outra.

O GWTP fornece uma forma prática de fazer isso, porém, somente uma das *Presenters* pode ser um Place. Na Figura 32 existem diversas *Presenters*, porém, apenas a *Presenter* indicada com o numeral “5” é um *Place*.

Figura 32 - NestedPresenters

Na Figura 32, basicamente, todas as cinco *Presenters* estão contidas em uma *Presenter* principal, a `MainPresenter`. O código dessa *Presenter* pode ser observado na Figura 33.

```
public class MainPresenter extends Presenter<MainView, MainProxy>
{
    @ContentSlot
    public static final Type<RevealContentHandler<?>> TYPE_SetTopoContent
        = new Type<RevealContentHandler<?>>();

    @ContentSlot
    public static final Type<RevealContentHandler<?>> TYPE_SetMainContent
        = new Type<RevealContentHandler<?>>();

    @ContentSlot
    public static final Type<RevealContentHandler<?>> TYPE_SetMenuContent
        = new Type<RevealContentHandler<?>>();

    @ProxyStandard
    public interface MainProxy extends Proxy<MainPresenter>
    {
    }

    public interface MainView extends View
    {
    }

    private final MainMenuPresenter mainmenu;
    private final TopoPresenter topo;

    @Inject
    public MainPresenter(final EventBus eventBus, final MainView view,
            final MainProxy proxy, final MainMenuPresenter mainmenu,
            final TopoPresenter topo)
    {
        super(eventBus, view, proxy);
        this.mainmenu = mainmenu;
        this.topo = topo;
```

Figura 33 - Trecho de código da MainPresenter.

Pode-se observar na Figura 33, que das linhas 20 a 27 são declarados vários atributos anotados com `@ContentSlot`. A anotação `@ContentSlot` é utilizada para definir um tipo para ser usado nas *Presenters* “filhas”, quando for necessário incluí-las dentro da *Presenter* principal (GOOGLE CODE, 200-b). Ou seja, esses slots serão usados como uma espécie de ponto de referência para as *Presenters* filhas que poderão ser adicionadas a esta *Presenter* principal.

Outra particularidade que pode ser notada na Figura 33, é que o *Proxy* da *Presenter* possui a anotação `@ProxyStandard`. Esta anotação se faz necessária porque ela informa o GWTP que o *Proxy* dessa *Presenter* deve ser criado pelo *ProxyGenerator* do GWTP. *Standard* significa que não precisaremos dividir o código desta *Presenter*.

Ainda na Figura 33, é possível perceber que o construtor está injetando as instâncias de `MainMenuPresenter` e `TopoPresenter`. Na Figura 34, pode-se perceber que na implementação do método `onReveal`, são invocadas várias vezes o método `setInSlot` da superclasse `PresenterWidget`.

```
@Override
protected void onReveal()
{
    super.onReveal();
    setInSlot(TYPE_SetMenuContent, mainmenu);
    setInSlot(TYPE_SetTopoContent, topo);
}
```

Figura 34 - Implementação do método onReveal na classe MainPresenter.

É ainda, necessário que a implementação da *View* sobrescreva o método `setInSlot` da classe abstrata `ViewImpl`. Quando o método `setInSlot` da `MainPresenter` é invocado, o GWTP invoca também o método `setInSlot` sobrescrito na classe `MainView`, ficando ele responsável apenas por decidir qual componente irá representar cada slot definido na `MainPresenter`. Pode-se observar o código da implementação desse método na Figura 35.

```
@Override
public void setInSlot(Object slot, Widget content)
{
    if(slot == MainPresenter.TYPE_SetMainContent)
    {
        setMainContent(content);
    }
    else if(slot == MainPresenter.TYPE_SetMenuContent)
    {
        setMenuContent(content);
    }
    else if(slot == MainPresenter.TYPE_SetTopoContent)
    {
        setTopoContent(content);
    }
    else
    {
        super.setInSlot(slot, content);
    }
}
```

Figura 35 - Implementação do método setInSlot da MainView.

Após isso, o GWTP saberá como tratar essas *Presenters*, e, como pode ser observado na Figura 34, assim que a `MainPresenter` for revelada, ela exibirá também suas "*Presenters* filhas".

Neste exemplo, a `HomePresenter` também possui seus *Slots* (itens 4 e 5 da Figura 32). Estes, porém, são setados de forma dinâmica, alterando-se o método `revealInParent`, como pode ser observado na Figura 36, ao invés de substituir toda a página, substituímos apenas o conteúdo do *slot* `TYPE_SetQuadroScrumContent`. Isso permite que tenhamos diversas *Presenters* aninhadas de forma dinâmica, deixando o código ainda mais modular, simples e limpo.

```
@Override
protected void revealInParent()
{
    RevealContentEvent.fire(this,
            HomePresenter.TYPE_SetQuadroScrumContent, this);
}
```

Figura 36 - Implementação do método revealInParent da classe QuadroScrumPresenter.

2.6.2 VIEW

Seguindo o modelo MVP, as `Views` do GWTP são simples e praticamente não possuem lógica. Seu único papel básico é implementar a interface relativa a sua `Presenter`, retornando os objetos necessários à ela.

A criação do *layout* da tela pode ser feita utilizando-se uma sintaxe muito semelhante à outras tecnologias (como o *Swing*), como já foi dito anteriormente.

A partir da versão 2.0 do GWT, tornou-se possível criar *layouts* de forma declarativa, utilizando arquivos XML, utilizando uma tecnologia denominada *UiBinder*.

No fundo, uma aplicação GWT é uma página *Web*. O modo mais natural de se desenhar uma página *Web*, é escrevendo HTML e CSS. O *framework UiBinder* permite que o desenvolvedor escreva suas aplicações como páginas HTML, com componentes espalhados por elas (GOOGLE CODE, 2009).

Além do fato de ser mais natural escrever a interface com o usuário utilizando *UiBinder* que em forma de código, o *UiBinder* também pode tornar a aplicação mais rápida.

Isso se deve ao fato de que, segundo o GOOGLE CODE (2009), os navegadores são melhores na construção de estruturas DOM inserindo várias grandes *Strings* de HTML em atributos *innerHTML* do que através de diversas chamadas à API. O UiBinder tira vantagem disso, e o resultado disso é a maneira mais agradável de construir a aplicação, é também, a melhor maneira de construí-la (GOOGLE CODE, 2009).

As principais vantagens de se utilizar o *UiBinder* são:

- Aumenta a produtividade e manutenabilidade da aplicação;
- Facilita a colaboração de designers de interfaces com o usuário, que, no geral, ficam mais confortáveis trabalhando com XML, HTML e CSS do que com código Java;
- Fornece uma transição gradual durante o desenvolvimento de protótipos em HTML para interfaces reais e interativas;
- Encoraja uma separação clara da estética (um modelo com XML) e do seu comportamento programático (uma classe Java);
- Executa verificações em tempo de compilação de referências cruzadas a partir do código Java para o XML e vice-versa;
- Oferece suporte a internacionalização;
- Incentiva o uso mais eficiente dos recursos do navegador, tornando mais conveniente usar elementos HTML leves ao invés de componentes pesados.

O *UiBinder* **não** é um renderizador. Não existem *loops* ou condicionais. É apenas uma versão limitada de uma linguagem de expressão.

A utilização do *UiBinder* é tão simples quanto criar uma interface e um arquivo XML, ambos no mesmo pacote e com o mesmo nome, se ignorada a extensão (por exemplo, `LoginViewImpl.java` e `LoginViewImpl.ui.xml`).

Na Figura 37 pode-se ver que, além dos componentes padrões, é possível importar outros pacotes, sejam eles de componentes ou qualquer outra coisa que possa auxiliar o desenvolvedor na criação da tela. No exemplo da Figura 37, pode-se notar que está sendo importado um módulo de internacionalização, por exemplo.

```
<!DOCTYPE ui:UiBinder SYSTEM "http://dl.google.com/gwt/DTD/xhtml.ent">
<ui:UiBinder xmlns:ui="urn:ui:com.google.gwt.uibinder"
    xmlns:g="urn:import:com.google.gwt.user.client.ui">
    <ui:with type="com.geekvigarista.scrummanager.client.i18n.Mensagem"
        field="msg"></ui:with>
    <g:HTMLPanel ui:field="conteudo">
        <g:VerticalPanel width="100%" height="300px"
            horizontalAlignment="ALIGN_CENTER" verticalAlignment="ALIGN_MIDDLE" ui:fiel
            <g:DecoratorPanel>
                <g:VerticalPanel>
                    <g:Image width="316px" height="91px" ui:field="logo" />
                    <g:Label text="{msg.efetuarlogin}" styleName="titulo1" direction="F
                    <g:HorizontalPanel>
                        <g:Grid width="351px" height="113px">
                        <g:row>
                            <g:customCell>
                                <g:Label text="{msg.usuario}" />
                            </g:customCell>
                            <g:customCell>
                                <g:TextBox ui:field="login" />
                            </g:customCell>
                        </g:row>
                        <g:row>
                            <g:customCell>
                                <g:Label text="{msg.senha}" />
                            </g:customCell>
```

Figura 37 - Trecho de código do arquivo XML de uma interface construída de forma declarativa.

Pode-se observar também na Figura 37 que os componentes não aceitam muitos parâmetros ao utilizá-los de forma declarativa. Este é o motivo do *UiBinder* necessitar também de uma classe Java. Essa classe será a responsável por permitir que o desenvolvedor possa obter os dados de um campo, por exemplo. Porém, para que isso se torne possível, é necessário que o componente tenha recebido um valor para o parâmetro `ui:field`, e que a classe Java contenha um atributo do mesmo tipo e com o mesmo nome que foi definido na propriedade `ui:field`, anotado com `@UiField`. Pode-se observar isso na Figura 37 e Figura 38.

Ainda na Figura 38, nota-se a criação da interface `LoginViewImplUiBinder`. Essa interface faz o papel de “ponte” entre o arquivo XML e a classe Java. Também é possível perceber que é utilizada a *factory* padrão do GWT para obter uma instância dessa interface, e que, após isso, a classe `LoginViewImpl` está apta à utilizar todos os componentes declarados no arquivo XML.

```
public class LoginViewImpl extends ViewImpl implements LoginPresenter.LoginView, Resi
{
    private static LoginViewImplUiBinder uiBinder = GWT.create(LoginViewImplUiBinder.
    interface LoginViewImplUiBinder extends UiBinder<Widget, LoginViewImpl>{}

    @UiField HTMLPanel conteudo;
    @UiField TextBox login;
    @UiField TextBox passwd;
    @UiField Button btlogin;
    @UiField CheckBox lembrar;
    @UiField VerticalPanel vpTudo;
    @UiField Image logo;

    public LoginViewImpl()
    {
        uiBinder.createAndBindUi(this);
        vpTudo.setHeight(Window.getClientHeight() + "px");
        logo.setUrl(Images.instance.logo().getSafeUri());
        Window.addResizeHandler(this);
    }

    @Override
    public Widget asWidget()
    {
        return conteudo;
    }

    @Override
    public TextBox login()
    {
        return login;
    }
```

Figura 38 – Trecho de código da implementação de uma View utilizando UiBinder.

2.6.3 PLACE MANAGER

A *History* API do GWT permite que o desenvolvedor possa atrelar uma *Presenter* a um *token* de uma URL. Porém, a tarefa de implementar uma versão funcional à partir da API do padrão do GWT acaba se tornando morosa, e o código acaba tornando-se repetitivo e de difícil manutenção. Visando facilitar isso, o GWTP implementou uma solução chamada `PlaceManager`.

O `PlaceManager` funciona como um intermediário entre os `ProxyPlaces` e a *History* API do GWT. É chamado de ProxyPlace qualquer interface `Proxy` de qualquer *Presenter* que estenda `ProxyPlace`. Ao criar um *Proxy* estendendo `ProxyPlace`, torna-se obrigatório o uso da anotação `@NameToken` na interface

Proxy em questão, para que o GWTP possa ser capaz de saber que determinado *token* pertence a determinada *Presenter*. Também se faz necessária a implementação de uma classe estendendo `PlaceManagerImpl` (Figura 39). Essa classe será responsável por tratar eventos irregulares relacionados à URLs de acesso, como tentativa de acesso a *tokens* inválidos e/ou não autorizados.

```
public class ClientPlaceManager extends PlaceManagerImpl
{
    private final PlaceRequest defaultPlaceRequest;

    @Inject
    public ClientPlaceManager(EventBus eventBus, TokenFormatter tokenFormatter, @DefaultPlace Strir
    {
        super(eventBus, tokenFormatter);
        this.defaultPlaceRequest = new PlaceRequest(defaultNameToken);
    }

    @Override
    public void revealDefaultPlace()
    {
        revealPlace(defaultPlaceRequest, false);
    }

    @Override
    public void revealErrorPlace(String invalidHistoryToken)
    {
        System.out.println("Token inválido: " + invalidHistoryToken);
        revealPlace(new PlaceRequest(NameTokens.erro404));
    }

    @Override
    public void revealUnauthorizedPlace(String unauthorizedHistoryToken)
    {
        PlaceRequest pr = new PlaceRequest(NameTokens.login).with(Parameters.u, unauthorizedHistory
        revealPlace(pr);
    }
```

Figura 39 - Exemplo de implementação da classe PlaceManagerImpl.

Através do `PlaceManager`, torna-se possível utilizar URL *Parameters*. Os principais objetivos de se usar URL *Parameters* são salvar parte do estado de uma página na URL (suporte a histórico de navegação) e tornar a página capaz de ser indexada por buscadores. Os URL *Parameters* também são utilizados para facilitarem o *bookmarking* de páginas específicas da aplicação, e, principalmente, facilitarem a navegação entre páginas distintas. Na Figura 40 é possível observar um exemplo de URL que utiliza *Parameters* com o GWT.

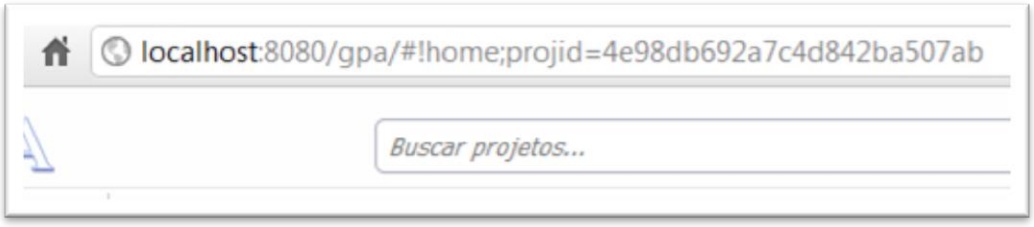


Figura 40 – Exemplo de URL Parameter.

Na URL mostrada na Figura 40, nota-se a presença de um conjunto de caracteres denominado *hashbang* (#!). *Hashbangs* são utilizados para indicar ao *crawler* do *Google* que a URL em questão é indexável, diferentemente de URLs que contenham somente o caractere # (utilizado para indicar ao navegador que ele deve se posicionar em uma `div` ou título específico dentro da página atual). Os *Hashbangs* estão sendo amplamente utilizados por diversas aplicações *Web* modernas, como o *Twitter* e o *Facebook* para fazer uso das vantagens anteriormente citadas.

O estado de uma página pode ser facilmente recuperado pela aplicação, desde que a `Presenter` seja um `ProxyPlace` e que ela implemente corretamente o método `prepareFromRequest`.

```
@Override
public void prepareFromRequest(PlaceRequest request)
{
    super.prepareFromRequest(request);
    final String id = request.getParameter(Parameters.projid, null);

    if(id == null)
    {
        setProjeto(null);
        return;
    }

    new AbstractCallback<LoadProjetoResult>()
    {
        @Override
        protected void callService(AsyncCallback<LoadProjetoResult> asyncCallback
        {
            dispatch.execute(new LoadProjetoAction(id), asyncCallback);
        }

        @Override
        public void onSuccess(LoadProjetoResult result)
        {
            setProjeto(result.getProjeto());
        }
    }.goDefault();
}
```

Figura 41 - Trecho de código de um exemplo de implementação do método prepareFromRequest

Na Figura 41 pode ser observado que o método `prepareFromRequest` recebe por parâmetro um objeto do tipo `PlaceRequest`. O desenvolvedor torna-se capaz então de capturar os valores de todos os atributos recebidos, como o código (*id*) de algum objeto persistido em um bando de dados, por exemplo, que pode ser recuperado posteriormente através de chamadas assíncronas para o servidor da aplicação.

2.6.4 GATEKEEPER

O `Gatekeeper` é uma funcionalidade do GWTP que tem como objetivo básico, tornar fácil para o desenvolvedor bloquear a exibição de determinadas `Presenter` casos pré-definidos por ele, como, por exemplo, páginas que só podem ser acessadas por um usuário logado ou por um usuário com permissão de administrador (exemplo da Figura 42).

Utilizá-lo é bastante simples, basta criar uma classe que implemente `Gatekeeper` e configurá-la no GIN (conforme já foi explicado anteriormente), para que ela possa ser injetada nas `Presenters` que precisem dela.

```
@Singleton
public class AdminGatekeeper implements Gatekeeper
{

    private final UsuarioLogadoGatekeeper usuarioGatekeeper;

    @Inject
    public AdminGatekeeper(UsuarioLogadoGatekeeper usuarioGatekeeper)
    {
        super();
        this.usuarioGatekeeper = usuarioGatekeeper;
    }

    @Override
    public boolean canReveal()
    {
        return usuarioGatekeeper != null && usuarioGatekeeper.getUsuario() !=
    }

}
```

Figura 42 - Exemplo de Gatekeeper.

Para que uma `Presenter` possa ser exibida somente sob o consenso de um `Gatekeeper`, deve-se anotar o `Proxy` dela com `@UseGatekeeper`, passando como parâmetro a classe do `Gatekeeper` responsável. Após isso, o `PlaceManager` garantirá que a `Presenter` só será exibida caso o método `canReveal` de seu `Gatekeeper` retorne **`true`**, e, caso contrário, o usuário será redirecionado para uma página de erro, de acesso negado, ou mesmo para a `Presenter` padrão da aplicação. O comportamento a respeito de o que o `PlaceManager` deve fazer nesses casos pode ser alterado sobrescrevendo o método `revealUnauthorizedPlace` na classe do `PlaceManager` da aplicação.

2.6.5 COMMAND PATTERN REUTILIZÁVEL COM GWT-DISPATCHER

O *Command* é um dos 11 padrões de projeto comportamentais do Catálogo GoF, e tem por objetivo básico encapsular uma solicitação em um objeto. Isso

permite que clientes parametrizem, enfileirem ou façam log de diversas solicitações, e ainda, permite que operações possam ser facilmente desfeitas ou refeitas.

O GWTP herdou uma implementação reutilizável desse padrão do *GWT-Dispatch*. Utilizando-o, temos as operações envolvidas em classes, que são enviadas para o servidor e retornam na forma de respostas, também envolvidas em classes. Esse “transporte” ocorre através de um “despachante”, que é a origem desse nome.

Para fazer uso do *Dispatch*, precisa-se basicamente de uma classe que encapsule uma ação, uma que encapsule o resultado dessa ação, e ainda uma terceira, que será a classe responsável por receber a ação encapsulada, executar o que for necessário, e retornar a resposta encapsulada na classe de resultado. Essa classe é comumente chamada de `ActionHandler`.

Começaremos por uma classe de resultado. Supondo uma ação simples, como salvar usuário no banco de dados através de um DAO. Conforme pode ser visto na Figura 43, a classe de resultado é bastante simples. Ele somente encapsula as propriedades que necessitamos retornar ao servidor, e, obrigatoriamente, implementa a interface `Result`. Também é necessário ter um construtor vazio, pois, como instâncias desse objeto irão trafegar do servidor para o cliente, elas precisam ser serializáveis, para que possam ser convertidas para *JavaScript*.

```
public class SalvarUsuarioResult implements Result {
    private Usuario response;

    public SalvarUsuarioResult(Usuario response) {
        super();
        this.response = response;
    }

    @SuppressWarnings("unused")
    private SalvarUsuarioResult() {
        // somente para serialização...
    }

    public Usuario getResponse() {
        return response;
    }
}
```

Figura 43 - Exemplo de classe de Resultado.

A classe que encapsula a ação (Figura 44), geralmente, é tão simples quanto a de resultado. Essa classe costuma ser bastante parecida com a classe de resultado, divergindo dela apenas quanto ao fato de implementar `ActionImpl` ou `UnsecureActionImpl` ao invés de Result.

A diferença básica entre se utilizar `ActionImpl` ou `UnsecureActionImpl` é que utilizando `UnsecureActionImpl`, as requisições não são protegidas contra ataques XSRF (*Cross-Site Request Forgery*) [7]. Usar `ActionImpl`, porém, obriga o desenvolvedor a implementar um `SecurityCookie`[8], que é responsável por proteger as requisições contra esses e possivelmente também contra outros tipos de ataques.

[7] XSRF ou CSRF é um tipo de ataque que explora a confiança que o site tem no usuário. Geralmente o ataque é feito utilizando-se de *exploits* maliciosos que executam comandos em determinados sites sem o conhecimento do usuário.

[8] O tutorial oficial de configuração de um filtro de segurança contra ataques XSRF pode ser encontrado na *WIKI* oficial do *GWT-Platform*, disponível em http://code.google.com/p/gwt-platform/wiki/GettingStartedDispatch#Protecting_against_XSRF_attacks.

```
public class SalvarUsuarioAction extends UnsecuredActionImpl<SalvarUsuarioResult> {

    private Usuario usuario;

    public SalvarUsuarioAction(Usuario usuario) {
        super();
        this.usuario = usuario;
    }

    @SuppressWarnings("unused")
    private SalvarUsuarioAction() {
    }

    public Usuario getUsuario() {
        return usuario;
    }
}
```

Figura 44 - Exemplo de classe encapsulando uma ação.

Assim como as classes de resultado, as classes de ação também irão trafegar pela rede, então, também precisam de um construtor vazio, pelos mesmos motivos citados anteriormente.

A terceira e última classe necessária, é a que fica somente no lado servidor da aplicação. Usualmente, ela ira importar um controlador ou até mesmo diretamente um DAO, efetuar as lógicas necessárias, montar um resultado e retorná-lo. Seguindo a linha de pensamento dos exemplos anteriores, na Figura 45 é possível ver um trecho de código correspondente à implementação de um `ActionHandler` simples para salvar um usuário.

```
public class SalvarUsuarioActionHandler
    implements ActionHandler<SalvarUsuarioAction, SalvarUsuarioResult>
{

    Injector inj = Guice.createInjector(new DAOModule());

    IDaoUsuario dao = inj.getInstance(DaoUsuario.class);

    @Override
    public SalvarUsuarioResult execute(SalvarUsuarioAction arg0,
            ExecutionContext arg1) throws ActionException
    {
        Usuario u = arg0.getUsuario();
        RetornoValidacao rv = valida(u);
        if(!rv.isOk())
        {
            return new SalvarUsuarioResult(rv.getErros());
        }

        dao.salvar(u);
        return new SalvarUsuarioResult(u);
    }
```

Figura 45 - Trecho de código exemplificando uma implementação de ActionHandler para salvar um usuário.

Observando-se a Figura 45, vemos que esta classe implementa `ActionHandler`, com um diamante esperando uma *Action* e um *Result* (que são as classes de ação e resultado implementadas anteriormente). A interface `ActionHandler` te obriga a implementar os métodos `execute`, `undo` e `getActionType`, que são, respectivamente, o método padrão a ser executado, o método chamado em uma ação de desfazer e o método que retorna o tipo da ação, que nada mais é que a classe da *Action*.

Após essas três classes implementadas e o *servlet* do *dispatch* devidamente configurado[9], é necessário dizer ao *Dispatch* qual `ActionHandler` é o responsável por cada ação. Para isso, é necessário adicionar um código semelhante ao da Figura 46 na classe configurada como módulo do servidor.

```
bindHandler(SalvarUsuarioAction.class, SalvarUsuarioActionHandler.class);
```

[9] O tutorial de configuração do *Dispatch* pode ser encontrado em http://code.google.com/p/gwt-platform/wiki/GettingStartedDispatch

Figura 46 - Bind do ActionHandler com a Action.

Após isso, o desenvolvedor torna-se apto a utilizar essa ação no lado cliente da aplicação, necessitando apenas de uma instância da interface `DispatchAsync`, que pode ser obtida através de injeção de dependência. Um exemplo básico de como fazer uma chamada assíncrona utilizando essa `ActionHandler` pode ser visto na Figura 47.

```
dispatcher.execute(new SalvarUsuarioAction(usuario),
        new AsyncCallback<SalvarUsuarioResult>()
{

    @Override
    public void onFailure(Throwable caught)
    {
        new MsgBox(caught.getMessage(), true);
    }

    @Override
    public void onSuccess(SalvarUsuarioResult result)
    {
        if(result.getErros() == null || result.getErros().isEmpty())
        {
            String msg = "Usuario "
                    + result.getResponse().getNome() + " salvo com sucesso";
            new MsgBox(msg, false);
            setUsuario(result.getResponse());
        }
        else
        {
            new MsgBox(result.getErros(), true);
        }
    }
});
```

Figura 47 - Exemplo de chamada de uma Action no lado cliente da aplicação.

Levando em conta os recursos da orientação à objetos e da linguagem Java, torna-se possível criar classes abstratas, por exemplo, que generalizam algumas ações, como a exibição de mensagens de erro padrões, alguma mensagem indicando que uma requisição está em andamento, ou ainda fazer com que a classe

automaticamente faça mais de uma tentativa de comunicação com o servidor em caso de falha[10].

É de extrema importância que o desenvolvedor realmente leve as chamadas assíncronas a sério. O desenvolvedor não pode, em momento algum, esquecer o que o A da sigla AJAX significa. Tudo pode exigir uma chamada assíncrona em algum momento, então, é melhor presumir que sempre será (RAY, 2009). Levando isso em conta, é bom que as classes que serão transmitidas nas chamadas também estejam preparadas para tal. A melhor forma de fazer isso é criando DTOs (*Data Transfer Object*), classes que serão utilizadas somente para o transporte de dados específicos de alguma requisição. As classes de retorno do GWT-Dispatch, se implementadas corretamente, suprem essa necessidade.

[10] Um exemplo de implementação com as funcionalidades descritas pode ser encontrado em http://code.google.com/p/gwt-scrum-manager/source/browse/trunk/src/com/geekvigarista/scrummanager/client/telas/commons/AbstractCallback.java

3 MATERIAL E MÉTODOS

A metodologia utilizada para o desenvolvimento do trabalho envolveu primeiramente uma revisão bibliográfica das linguagens e tecnologias aplicadas no desenvolvimento do estudo experimental. Após o estudo, foi então feita uma análise e uma pequena prototipação da aplicação proposta. Por fim, a aplicação foi codificada utilizando o ambiente de desenvolvimento descrito a seguir.

O sistema proposto neste trabalho foi desenvolvido em conjunto com Raduan Silva dos Santos, o qual implementou toda a parte Servidor da aplicação, implementando a camada de persistência e o banco de dados em si.

3.1 AMBIENTE DE DESENVOLVIMENTO

Para a codificação da aplicação foi utilizada o *Eclipse*, um dos principais Ambientes de Desenvolvimento Integrado (IDE – *Integrated Development Environment*) utilizado por milhões de desenvolvedores. O *Eclipse* é desenvolvido pela *Eclipse Foundation*, um consórcio de grandes empresas e membros da comunidade que financiam os projetos da fundação através de uma contribuição anual e que também ajudam a definir seu futuro, além de hospedar e incubar os vários projetos da ferramenta (ECLIPSE, 2009).

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-).

Foi também utilizado o *Google Code* como repositório SVN (*Subversion*), para que fosse possível trabalhar em conjunto e manter o código sincronizado facilmente entre os desenvolvedores. Para facilitar a sincronização, foi utilizado um plugin em conjunto com o *Eclipse* (*Subversive*).

3.2 ANÁLISE

O sistema tem como objetivo principal o cadastro e controle de projetos, seus requisitos e *stakeholders* envolvidos (todos os integrantes do projeto), assim como o controle de fluxo dos encaminhamentos de requisitos. O sistema terá 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-its*".

O quadro do *Scrum* apresenta os requisitos divididos em quatro colunas que representam seu estado. Os estados em que um requisito poderá estar serão: aguardando, em desenvolvimento, em testes e concluído.

Quando o requisito for criado, será criado automaticamente um encaminhamento padrão para aguardando, para que o requisito seja posicionado corretamente no quadro do projeto correspondente.

Quando um usuário encaminhar um requisito, alterando seu estado, um novo encaminhamento deverá ser criado. Os encaminhamentos devem seguir o fluxo do quadro, ou seja, não será possível encaminhar um requisito do estado "aguardando" diretamente para concluído.

Todo usuário que utilizará o sistema terá um *login* e uma senha para efetuar *login*. Usuários poderão ser diferentes *stakeholders* em diferentes projetos ou até mesmo em um mesmo projeto. Somente usuário com permissão de administrador poderão cadastrar novos projetos, usuários, produtos e *stakeholders*.

Todo projeto deve pertencer a um produto específico, visando manter uma melhor organização dos projetos futuramente.

Ao adicionar requisitos à um projeto, o sistema deverá automaticamente criar uma *Sprint* para ele. Uma *Sprint* não terá um tempo fixo, ficando a cargo do Gerente de Projetos adicionar os requisitos de acordo com o tempo que ele tem disponível.

Nesta primeira versão da aplicação, não serão tratados requisitos impactantes, S*print ba*cklogs, casos de uso e de testes entre outros, ficando essas funcionalidades a serem implementadas em projetos futuros.

A análise deste projeto possui diversos casos de uso, porém, expandir todos eles aqui tornaria este trabalho muito extenso, sendo assim, apenas um caso de uso de exemplo será mostrado de forma completa.

3.2.1 CASO DE USO MANTER USUÁRIOS

A cargo de exemplo, neste item será mostrado um caso de uso mais simples, o de manter usuários, que pode ser observado no Quadro 2. A listagem dos demais casos de uso pode ser observada no Apêndice A.

Caso de uso 1 – Manter Usuários

1. Descrição: O caso de uso de manutenção de usuário tem como função apresentar os passos necessários ao desenvolvimento do cadastro de usuários, levanto em conta também a edição de usuário.

2. Atores envolvidos: Usuário administrador.

3. Pré-condições: Usuário administrador logado no sistema.

4. Fluxo de eventos

    a. Fluxo Básico:
        i. O usuário clica no menu novo, em seguida no *submenu* “Usuário”
        ii. A tela de Cadastro de Usuários é mostrada ao usuário com os seguintes campos :
            1. Nome – Campo de texto para preenchimento do nome do usuário. Seu preenchimento é obrigatório;
            2. *Login* – Campo de texto para preenchimento do *login* do usuário, ou seja, o nome no qual ele usará para entrar no sistema. Seu preenchimento é obrigatório;
            3. Senha – Campo de texto oculto (próprio para senhas), para o preenchimento da senha, a qual o usuário usara

para entrar no sistema. Seu preenchimento é obrigatório. A senha obrigatoriamente deverá ter no mínimo seis caracteres;

4. Confirmação de Senha – Campo de texto oculto para o preenchimento da confirmação de senha. A confirmação de senha deve ser exatamente igual ao valor do campo senha. Seu preenchimento é obrigatório;
5. Administrador – Caixa de seleção para indicar se o usuário a ser cadastrado terá permissões de administrador, ou somente será um usuário comum.

iii. A tela de cadastro de usuário terá um botão pequeno, de tamanho suficiente para conter um caractere “+”.

1. O texto do botão será “+”.
2. A função do botão será cadastrar novos usuários. Ao clicar no botão “+”, ele deve limpar todos os campos da tela de cadastro de usuário, e os objetos usados na tela.
3. O botão de “+” deve ficar localizado no canto inferior esquerdo da tela, como visto na figura 9.

iv. A tela de cadastro de usuário terá também um link com o texto “cancelar”.

1. A função deste link é cancelar o cadastro de usuário, limpar os campos e objetos utilizados para aquele cadastro e fechar a tela.
2. Ao clicado no botão, ele deve limpar os objetos utilizados e enviar para a tela inicial do sistema.
3. O link de cancelar deve ficar localizado no canto inferior direito da tela, como pode ser visto na figura 9.

v. A tela de cadastro de usuário terá também um botão com o texto “salvar”.

1. O botão salvar terá a função de salvar os dados da tela em objeto no banco.
2. Ao clicar em salvar, o botão deverá chamar as funções responsáveis por salvar usuário.

3. O botão salvar deve ficar localizado no canto inferior direito da tela, a direita do link cancelar.

vi. A função de salvar deverá validar se todos os campos estão preenchidos:

1. Caso todos os campos estejam preenchidos, deverá enviar para os métodos responsáveis por salvar o objeto Usuário, tendo o objeto preenchido com os valores correspondentes aos campos da tela.
2. Caso algum campo não esteja preenchido, deve retornar uma ou várias mensagens de erro na tela com a informação sobre quais campos estão sem preenchimento.

b. Fluxos de Exceção

i. O usuário clica no botão salvar com um ou mais campos sem preencher.

1. O sistema verifica que um ou mais campos precisam ser preenchidos.
2. O sistema mostra uma ou mais mensagens de erro, uma para cada campo que precisa ser preenchido.

ii. O usuário preenche o nome ou *login* de um usuário já cadastrado no sistema.

1. O sistema verifica que o usuário com o nome ou *login* correspondente já esta cadastrado no sistema.
2. O sistema mostra a mensagem de erro correspondente.

iii. O usuário clica em salvar com os campos preenchidos, mas com a senha com menos de seis caracteres.

1. O sistema verifica que a senha tem menos que seis caracteres.

Quadro 2 - Caso de Uso 1 - Manter Usuários.

O Diagrama de Casos de Uso para o caso de uso expandido pode ser visto na Figura 48.

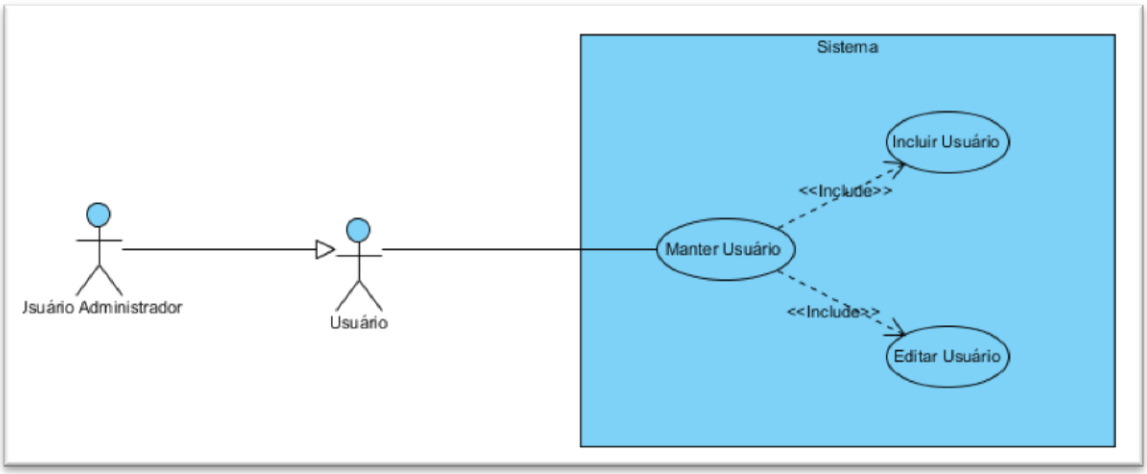


Figura 48 - Diagrama de Caso de Uso "Manter Usuários"

3.2.2 DIAGRAMA DE CLASSES

O diagrama de classes completo da aplicação ficaria ficou grande e complexo, o que impossibilita mostra-lo aqui, pois ficaria ilegível (a listagem das classes do dos pacotes `client` e `shared` da aplicação podem ser encontrados no Apêndice B). Sendo assim, será mostrado somente o diagrama de classes da tela (a *presenter* e a *view*), que por sua vez, pode ser observado na Figura 49.

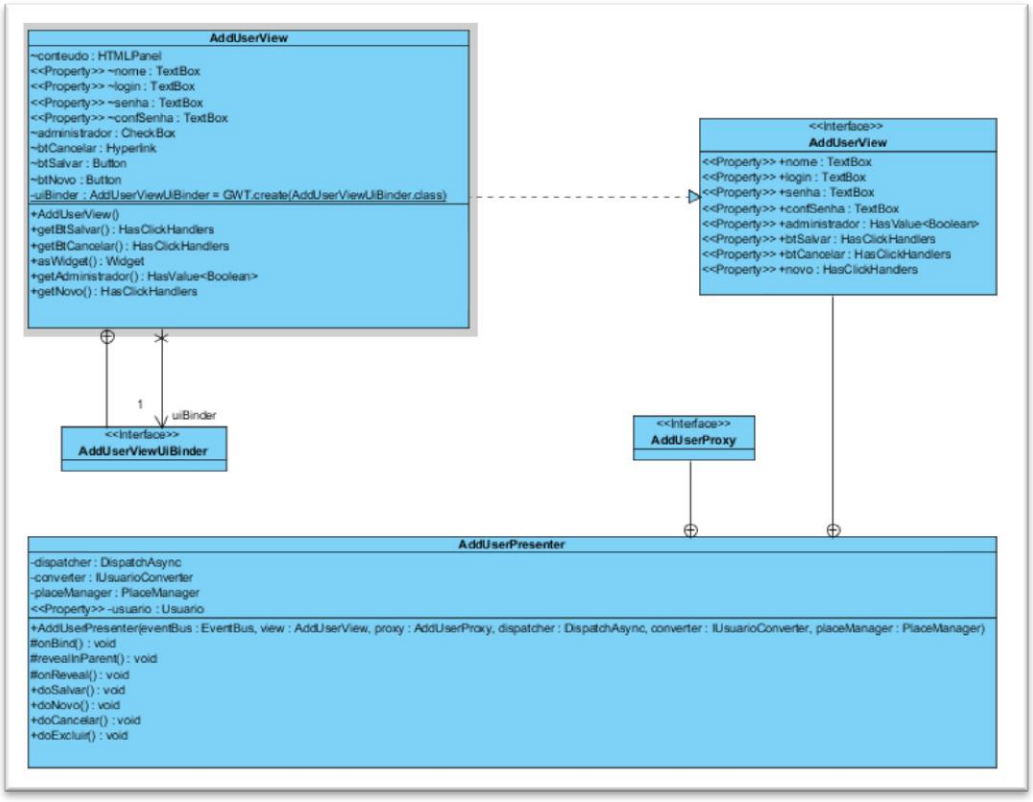


Figura 49 - Diagrama de Classes Básico da tela de Cadastro de Usuários.

Como pode-se ver, na Figura 49 existem duas classes principais (`AddUserPresenter` e `AddUserView`). A Presenter possui duas *inner interfaces*, chamadas `AddUserProxy` e `AddUserView`. Nota-se também que classe `AddUserView` implementa a *interface* `AddUserView` da Presenter, e que a classe `AddUserView` possui uma *inner interface* (`AddUserViewUiBinder`) utilizada pelo `UiBinder` para o desenho declarativo do *layout* da tela.

4 RESULTADOS E DISCUSSÕES

Este capítulo trata dos resultados alcançados durante o desenvolvimento do trabalho, sendo os principais pontos o desempenho e o baixo acoplamento. Serão também mostradas algumas telas da aplicação, e por fim, algumas vantagens, desvantagens e dificuldades encontradas ao longo do desenvolvimento.

Um dos focos da aplicação foi o altíssimo desempenho, e sendo esse item levado em consideração, juntamente com várias opiniões em listas de discussão e fóruns, foi decidido que as tecnologias a serem utilizadas seriam, principalmente, GWT para as telas e comunicação com o servidor, e o *MongoDB* como banco de dados. Após uma série de pesquisas e testes, foi decidido também que seria utilizado o *framework GWT-Platform*, pelo fato de ele implementar diversas boas práticas de arquitetura de *software* e fornecer uma ótima base para o desenvolvimento ágil da aplicação, como já foi tratado ao longo deste trabalho.

O desempenho obtido pode ser considerado bom. Na Figura 50 pode ser observado o tempo necessário e a quantidade de dados trafegados na rede para a ação de selecionar um projeto (esta ação carrega e exibe todos os requisitos e seus estados no quadro). A ação levou apenas 27ms, e transferiu algo em torno de 5KB (envio e retorno dos dados). Tanto o tempo como a quantidade de dados transferidos podem ser considerados pequenos, levando em conta que a lógica para a construção do quadro pode ser um pouco lenta e que era necessário uma quantia considerável de informações para concluir esta ação.

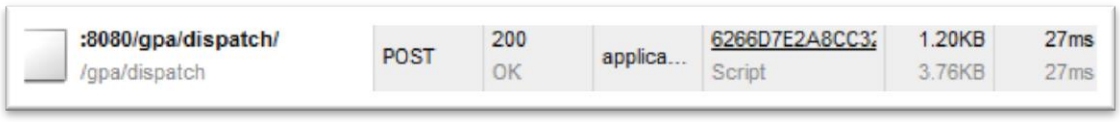


Figura 50 - Tempo levado para selecionar um projeto e carregar seus requisitos.

Outros tempos obtidos podem ser vistos no Quadro 3, onde pode-se perceber que a ação mais lenta foi o carregamento da tela de *login*. Essa demora é pode ser facilmente explica porque, antes de exibir a tela de login, a aplicação

instancia várias coisas, como o injetor do GIN, os DAOs e outras coisas necessárias pela aplicação, que então ficam carregadas na memória durante a execução dos outros testes.

Ação	Tempo decorrido
Carregar tela de login (carregamento inicial da aplicação)	230ms
Logar na aplicação	12ms
Selecionar um projeto, carregar seus requisitos e exibi-los no quadro	27ms
Salvar um usuário	24ms
Abertura cadastro de requisitos (tela mais carregada da aplicação)	1.82s

Quadro 3 - Alguns tempos médios de execução de algumas ações na aplicação.

Seguindo diversos padrões providos pelo GWTP e pelo MVP, e, graças a diversas tecnologias aplicadas pelo GWT, a aplicação, além de rápida, ficou bastante leve. Como pode ser observado na Figura 51, a Ferramenta de Desenvolvedor do *Google Chrome* mostra um consumo de apenas 4,64MB de memória após alguns minutos de uso da aplicação, o que pode ser considerado um valor baixo, visto que outras aplicações relativamente simples, como o Twitter[11], por exemplo, usam em torno de 13MB (Figura 52).

[11] http://twitter.com

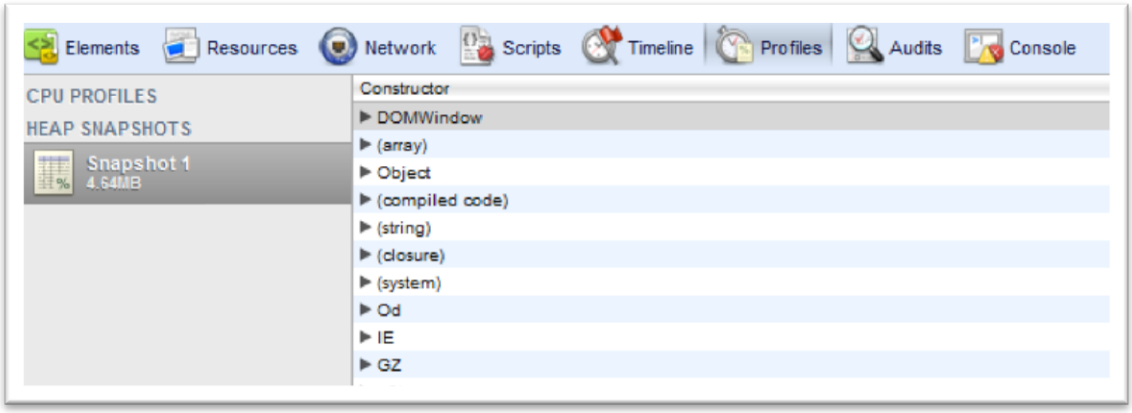


Figura 51 - Consumo de memória da aplicação desenvolvida, de acordo com a Ferramenta de Desenvolvedor do Google Chrome.

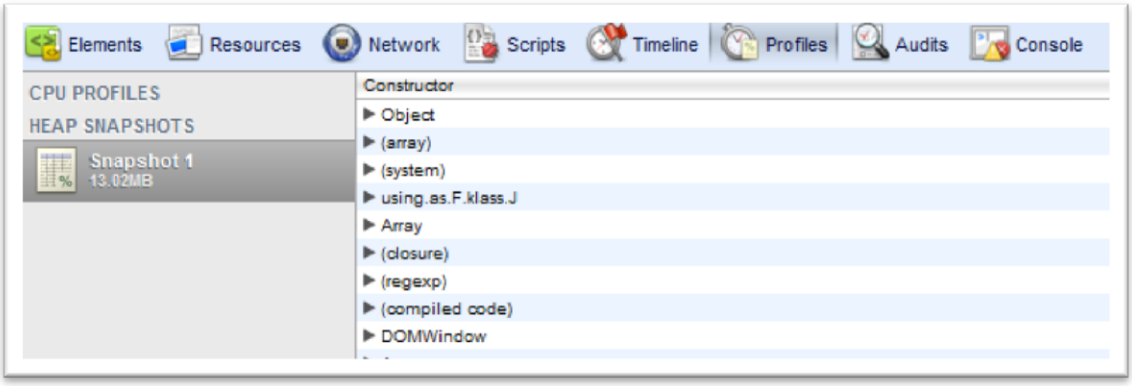


Figura 52 - Consumo de memória do Twitter, de acordo com a Ferramenta de Desenvolvedor do Google Chrome.

O baixo consumo de memória pode oferecer diferenças significativas principalmente em máquinas e navegadores antigos, que não possuem ou não gerenciam a memória das melhores formas possíveis. Para o usuário final da aplicação o baixo consumo de memória pode melhorar consideravelmente a fluidez da aplicação e diminuir travamentos no navegador, dependendo muito da máquina e navegador utilizados.

A principal vantagem da utilização da injeção de dependência juntamente com o GWTP foi a redução considerável do acoplamento da aplicação, e a sua fácil modularização (caso se torne necessário). Praticamente tudo é implementação de interfaces, e as instâncias dessas interfaces são injetadas pelo *Guice* ou GIN, o que torna bastante simples a implementação de uma nova tela, de um novo DAO, de um

novo controlador, ou qualquer outra funcionalidade da aplicação, sendo necessário apenas criar outra implementação para a interface, e alterar a classe do módulo que o GIN ou o Guice utilizam para prover a instância dela.

A Figura 53 mostra a tela principal da aplicação, já com um projeto selecionado no lado esquerdo. É possível ver a simulação do quadro do *Scrum* criada pela aplicação. Infelizmente, não houve tempo suficiente para a implementação de um recurso "*Drag'n'Drop*"[12], tornando necessária a utilização de botões para mover um requisito de um estado para outro.

Figura 53 - Tela Inicial da Aplicação já pronta.

Na Figura 54 é possível observar a tela de adição de requisitos à um projeto. Tem-se no lado esquerdo uma lista com os requisitos já adicionados, onde é possível selecionar algum destes para a edição ou exclusão.

[12] Drag'n'Drop: Em português significa "Arrastar e Soltar". É um recurso muito utilizado em diversas aplicações, porém, não muito simples de ser implementado.

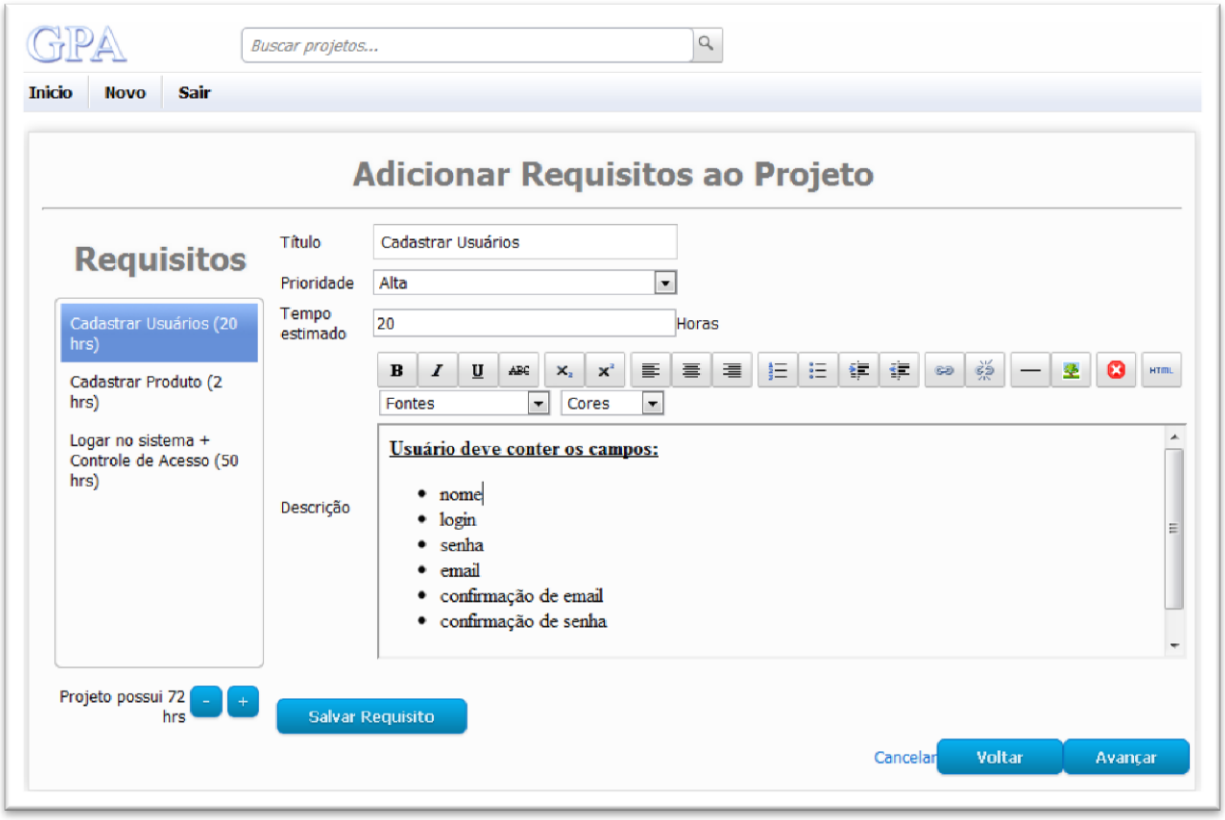


Figura 54 - Tela de adição de requisitos à um projeto.

O resultado final foi uma aplicação leve, rápida, bonita e de fácil manutenção, que, com algumas pequenas melhorias, pode se tornar uma aplicação amplamente utilizada para o gerenciamento de projetos de empresas ou times de universitários. Todo o código-fonte da aplicação pode ser encontrado no repositório SVN do GOOGLE CODE[13], juntamente com as bibliotecas necessárias para a execução da mesma. O arquivo `.war` da primeira versão, também pode ser encontrado na aba *downloads*[14] do repositório do GOOGLE CODE.

[13] O código-fonte da aplicação pode obtido através do endereço http://code.google.com/p/gwt-scrum-manager/

[14] O arquivo .war da primeira versão da aplicação pode ser obtido no endereço http://code.google.com/p/gwt-scrum-manager/downloads/

5 CONSIDERAÇÕES FINAIS

Este capítulo apresenta as conclusões obtidas durante a realização do trabalho, bem como sugestões para trabalhos futuros.

5.1 CONCLUSÃO

Este estudo abordou principalmente o funcionamento básico do GWT, o modelo MVP, o uso do *framework* GWTP juntamente com DI e *Command Pattern*, mostrando também exemplos de implementação baseados em uma aplicação real[15], desenvolvida paralelamente com este trabalho.

Com base no presente trabalho, pôde-se concluir que o *Scrum* é um *framework* para organização de equipes de desenvolvimento em geral. Embora o *Scrum* seja considerado bastante "liberal" quando comparado com outras metodologias de desenvolvimento, possui algumas filosofias que devem ser respeitadas pelas equipes. A aplicação desenvolvida ao longo desse trabalho não visa substituir de forma alguma o uso de um quadro físico em equipes em que os integrantes trabalham no mesmo local, mas sim, em equipes que trabalham distantes um do outro, nas quais geralmente a visibilidade do andamento do projeto e/ou *Sprint* torna-se difícil.

O GWT, apesar do pouco tempo de vida se comparado a outros *frameworks*, pode ser amplamente utilizado para o desenvolvimento de aplicações dinâmicas para a *Web* de forma ágil, levando em consideração que ele praticamente anula a necessidade da escrita de *JavaScript* pelo desenvolvedor, gerando o código *JavaScript* a partir de códigos Java, já otimizados para os principais navegadores. *Frameworks* para o GWT, como o GWTP (amplamente discutido nesse trabalho) tornam a tarefa de desenvolver com GWT ainda mais simples, tornando ainda o código mais fácil de manter e de dar manutenção. Uma das principais vantagens de se utilizar o GWT juntamente com o GWTP, é que o GWTP utiliza a DI de forma

[15] O código-fonte da aplicação pode obtido através do endereço http://code.google.com/p/gwt-scrum-manager/

bastante abrangente, o que torna o código bastante modular e de fácil compreensão para outros desenvolvedores.

O padrão MVC, embora ainda seja amplamente utilizado em vários *frameworks* e aplicações *Web*, segundo KEREKI (2010), não é um bom modelo para ser utilizado com o GWT, sendo o MVP o mais indicado para ser utilizado em seu lugar. O MVC é um modelo antigo, dá época em que a *Web* ainda não era popular, e nem sequer pensava-se em aplicações ricas para a Internet. O MVP foi então criado justamente para cobrir essa "falha" do MVC, e acabou conquistando vários adeptos e variações de implementação com as mais diversas tecnologias em todo o mundo.

Durante o desenvolvimento do trabalho e das pesquisas, foi possível perceber também que a maioria dos desenvolvedores *Web*, principalmente iniciantes, insistem em desenvolver aplicações AJAX sem implementar corretamente as chamadas assíncronas, seja transportando objetos muito pesados ou simplesmente tentando "forçar" as chamadas a serem síncronas, algo totalmente não recomendado.

A principal desvantagem encontrada foi a falta de documentação atualizada a respeito do GWTP. Esse fato acabou gerando algum atraso no desenvolvimento, pois, muitas vezes, a documentação estava desatualizada, o que tornava necessário a utilização de listas de discussão e fóruns como o *StackOverflow*[16] para a resolução de dúvidas conceituais e de detalhes da implementação.

5.2 TRABALHOS FUTUROS/CONTINUAÇÃO DO TRABALHO

O GWT possui diversas classes e APIs específicas para o desenvolvimento de testes unitários, tanto das lógicas de negócio, como das telas.

Testes unitários de telas em geral, costumam ser bastante complexos de ser implementados em sistemas *Web*, e encontra-se pouca documentação a respeito. A maioria dos *frameworks Web* não possuem APIs para o desenvolvimento de testes unitários em telas. O GWT, porém, devido, entre outros, ao fato de possuir uma ampla comunidade em torno de si, possui diversos frameworks e APIs que facilitam

[16] Perguntas sobre o GWTP no *StackOverflow* podem ser acessadas a partir do endereço http://stackoverflow.com/questions/tagged/gwt-platform

esse trabalho, como é o caso do *EasyMock*[17]. Uma sugestão para trabalhos posteriores seria um estudo aprofundado de alguma dessas APIs.

[17] A documentação e o *download* do *EasyMock* podem ser encontrados em http://easymock.org/.

APÊNDICE A – LISTA DE CASOS DE USO

Neste apêndice será mostrada a listagem de casos de uso da aplicação.

- Manter Usuário;
- Manter Stakeholder;
- Manter Projeto;
- Manter Produto;
- Manter Requisitos;
- Efetuar Login;
- Encaminhar Requisitos;
- Visualizar Status e Encaminhamento dos Requisitos;
- Exibir Quadro do Scrum.

APÊNDICE B – LISTAGEM DE CLASSES DA APLICAÇÃO

Neste apêndice serão listadas as classes e seus respectivos pacotes da aplicação, excluindo, porém, as *inner* classes e classes anônimas e as classes do pacote `server`.

- client
  - converters.interfaces.IProdutoConverter;
  - converters.interfaces.IProjetoConverter;
  - converters.interfaces.IRequisitoConverter;
  - converters.interfaces.IStakeholderConverter;
  - converters.interfaces.IUsuarioConverter;
  - converters.interfaces.ViewBeanConverter;
  - converters.ProdutoConverter;
  - converters.ProjetoConverter;
  - converters.RequisitoConverter;
  - converters.StakeholderConverter;
  - converters.UsuarioConverter;
  - events.LoginAuthenticatedEventHandler;
  - events.LoginAuthenticateEvent;
  - events.LogoutEvent;
  - events.LogoutEventHandler;
  - gatekeeper.AdminGatekeeper;
  - gatekeeper.UsuarioLogadoGatekeeper;
  - gin.ClientGinjector;
  - gin.ClientModule;

- gin.UtilsModule;
- gin.ViewBeanConverterModule;
- GWT_ScrumManager;
- i18n.Mensagem;
- i18n.MensagemParams;
- place.ClientPlaceManager;
- place.DefaultPlace;
- place.NameTokens;
- place.Parameters;
- telas.cadastros.interfaces.SimpleCadPresenter;
- telas.cadastros.produto.AddProdutoView;
- telas.cadastros.produto.CadastroProdutoPresenter;
- telas.cadastros.projeto.AddProjetoPresenter;
- telas.cadastros.projeto.AddProjetoView;
- telas.cadastros.requisito.AddRequisitoPresenter;
- telas.cadastros.requisito.AddRequisitoView;
- telas.cadastros.requisito.RequisitoItemsFactory;
- telas.cadastros.stakeholder.AddStakeholderPresenter;
- telas.cadastros.stakeholder.AddStakeholderView;
- telas.cadastros.stakproj.AddStakToProjPresenter;
- telas.cadastros.stakproj.AddStakToProjViewImpl;
- telas.cadastros.stakproj.StakeholderTableFactory;
- telas.cadastros.usuario.AddUserPresenter;
- telas.cadastros.usuario.AddUserView;
- telas.commons.AbstractCallback;

- telas.componentes.defaultbox.DefaultComponentsResources;
- telas.componentes.defaultbox.DefaultDateBox;
- telas.componentes.defaultbox.DefaultDialogBox;
- telas.componentes.defaultbox.DefaultIntegerBox;
- telas.componentes.defaultbox.DefaultListBox;
- telas.componentes.defaultbox.DefaultPasswordTextBox;
- telas.componentes.defaultbox.DefaultRichTextArea;
- telas.componentes.defaultbox.DefaultTextBox;
- telas.componentes.loading.events.LoadingStartEvent;
- telas.componentes.loading.events.LoadingStartEventHandler;
- telas.componentes.loading.events.LoadingStopEvent;
- telas.componentes.loading.events.LoadingStopEventHandler;
- telas.componentes.loading.IStatusPopupPanelHandler;
- telas.componentes.loading.LoadingResources;
- telas.componentes.loading.StatusPopupPanel;
- telas.componentes.loading.StatusPopupPanelHandler;
- telas.componentes.msgbox.MsgBox;
- telas.componentes.msgbox.MsgBoxResources;
- telas.componentes.richtexttoolbar.RichTextToolbar;
- telas.componentes.searchbox.events.SearchEvent;
- telas.componentes.searchbox.events.SearchEventHandler;
- telas.componentes.searchbox.HasSearchHandlers;
- telas.componentes.searchbox.Resources;
- telas.componentes.searchbox.SearchBox;
- telas.componentes.showmorepagepanel.ShowMorePagerPanel;

- telas.erros.Erro404ViewImpl;
- telas.erros.Error404Presenter;
- telas.inicio.events.abrirmodalencaminhar.AbrirModalEncaminharEvent;
- telas.inicio.events.abrirmodalencaminhar.AbrirModalEncaminharEventHandler;
- telas.inicio.events.encaminhar.EncaminharEvent;
- telas.inicio.events.encaminhar.EncaminharEventHandler;
- telas.inicio.events.projetoselecionado.ProjetoSelecionadoEvent;
- telas.inicio.events.projetoselecionado.ProjetoSelecionadoEventHandler;
- telas.inicio.events.updatesearchinput.UpdateSearchBoxValueEvent;
- telas.inicio.events.updatesearchinput.UpdateSearchBoxValueEventHandler;
- telas.inicio.home.HomePresenter;
- telas.inicio.home.HomeView;
- telas.inicio.home.projetos.ListaProjetosUsuarioPresenter;
- telas.inicio.home.projetos.ListaProjetosUsuarioView;
- telas.inicio.home.projetos.ProjetoCellFactory;
- telas.inicio.home.quadro.coluna.ColunaQuadroScrum;
- telas.inicio.home.quadro.modalencaminhar.ModalEncaminhar;
- telas.inicio.home.quadro.QuadroScrumPresenter;
- telas.inicio.home.quadro.QuadroScrumViewImpl;
- telas.inicio.home.quadro.requisitoquadro.RequisitoQuadro;
- telas.inicio.login.LoginPresenter;
- telas.inicio.login.LoginViewImpl;
- telas.inicio.main.MainPresenter;
- telas.inicio.main.MainView;

- telas.inicio.mainmenu.MainMenu;
- telas.inicio.mainmenu.MainMenuPresenter;
- telas.inicio.resultadobusca.ProjetoTableFactory;
- telas.inicio.resultadobusca.ResultadoBuscaPresenter;
- telas.inicio.resultadobusca.ResultadoBuscaViewImpl;
- telas.inicio.topo.TopoPresenter;
- telas.inicio.topo.TopoViewImpl;
- telas.interfaces.Images;
- telas.visao.requisito.EncaminhamentoAnteriorFactory;
- telas.visao.requisito.VisualizarRequisitoPresenter;
- telas.visao.requisito.VisualizarRequisitoView.

- shared
  - 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.

REFERÊNCIAS BIBLIOGRÁFICAS

BEAUDOIN, Philippe. **Google Web Toolkit and the Model View Presenter architecture**. In PrairieDevCon. 2011, Regina, Saskatchewan, Canada. Disponível em <http://gwt-platform.googlecode.com/files/GWT.pdf>. Acesso em: 18Out.2011.

COOPER, Robert T. et al; **GWT in Practice**, Manning Publications, 2008

ECLIPSE, **About Eclipse Foundation**. 2009. Disponível em <http://eclipse.org/org/>. Acesso em 03Nov.2011.

EGYEDI, Tineke M. **Why Java Was - Not – Standardized Twice**. 2001. Disponível em <http://csdl2.computer.org/comp/proceedings/hicss/2001/0981/05/09815015.pdf>. Acesso em 17Nov.2011.

Google Code, **Como funciona o Google Web Toolkit.** 200-a. Disponível em <http://code.google.com/intl/pt-BR/webtoolkit/overview.html>. Acesso em 05.Ago.2011.

Google Code, **GWTP (goo-teepee)**. 200-b. Disponível em <http://code.google.com/p/gwt-platform/>. Acesso em 27.Set.2011.

Google Code, **Guice**. 200-c. Disponível em <http://code.google.com/p/google-guice/>. Acesso em 28Set.2011.

Google Code, **Google-Gin**. 200-d. Disponível em <http://code.google.com/p/google-gin>. Acesso em 28Set.2011.

Google Code, **GinTutorial - Using GIN to create a GWT widget**. 2008. Disponível em <http://code.google.com/p/google-gin/wiki/GinTutorial>. Acesso em 27Set.2011.

Google Code, **Testing Methodologies Using Google Web Toolkit**. 200-e. Disponível em <http://code.google.com/intl/pt-BR/webtoolkit/articles/testing_methodologies_using_gwt.html>. Acesso em 26Set.2011.

Google Code, **Implements a reusable 'command pattern' API for GWT**. 200-f. Disponível em <http://code.google.com/p/gwt-dispatch/>. Acesso em 18Out.2011.

Google Code, **Implementation of the Presenter section of a Model-View-Presenter (MVP) pattern**. 200-g. Disponível em <http://code.google.com/p/gwt-presenter/>. Acesso em 18Out.2011.

Google Code, **Declarative Layout with UiBinder**. 2009. Disponível em <http://code.google.com/intl/pt-BR/webtoolkit/doc/latest/DevGuideUiBinder.html>. Acesso em 19Out.2011.

Google Code, **Developer's Guide - Internationalization**. 200-h. Disponível em <http://code.google.com/intl/pt-BR/webtoolkit/doc/latest/DevGuideI18n.html>. Acesso em 16Out.2011.

Google Code, **What's New in GWT 2.4?** 2011. Disponível em <http://code.google.com/intl/pt-BR/webtoolkit/doc/latest/ReleaseNotes.html>. Acesso em 29Out.2011.

HANSON, Robert et al; **GWT in action – Easy AJAX with Google Web Toolkit**, Manning Publications, 2007.

JAVA COMMUNITY PROCESS. 2005. Disponível em < http://jcp.org/en/jsr/detail?id=270>. Acesso em 03Nov.2011.

LINDHOLM, Tim et al. **The Java Virtual Machine Specification.** 1999. Disponível em <http://java.sun.com/docs/books/vmspec/download/vmspec.2nded.html.zip>. Acesso em 03Nov.2011.

ORACLE, **Java SE Technologies at a Glance**. 200-. Disponível em: <http://www.oracle.com/technetwork/java/javase/tech/technologies-135120.html>. Acesso em 02Nov.2011.

PRASANNA, Dhanji; **Dependency Injection**, Manning Publications, 2009.

PRESSMAN, Roger S.; **Software Engineering: A Practitioner's Approach, McGraw Hill**, 2001.

RAY, Ryan, **Google Web Toolkit Architecture: Best Practices For Architecting Your GWT App**. In Google I/O, 2009. San Francisco, California. Disponível em <http://www.google.com/intl/pt-BR/events/io/2009/sessions/GoogleWebToolkitBestPractices.html>. Acesso em 18Out.2011.

SANTOS, Rildo F. **SCRUM Experience**. 200-. Disponível em <http://www.etecnologia.com.br/scrum/Scrum%20Experience%20%5BO%20Tutorial%20SCRUM%5D%20v16.pdf>. Acesso em 15Out.2011.

SCHWABER, Ken. **Scrum Guide**. 2011 Disponível em <http://www.scrum.org/storage/scrumguides/Scrum_Guide.pdf>. Acesso em 14Out.2011.

Scrum Alliance, **What is Scrum?** 200-. Disponível em <http://www.scrumalliance.org/pages/what_is_scrum>. Acesso em 04Ago.2011.

KEREKI, Federico; **Essential GWT: Building for the Web with Google Web Toolkit 2 (Developer's Library)**, Addison Wesley, 2010.

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 02Set.2011.

W3C, **XHTML2 Working Group Home Page**. 2000. Disponível em <http://www.w3.org/MarkUp/>. Acesso em 02Ago.2011.

W3C Brasil, **Curso de HTML5.** 2010. Disponível em <http://www.w3c.br/cursos/html5/>. Acesso em 02Ago.2011.