Manual gluemvc 0.1.x

Português (BR)

Janeiro/2006

Sumário

Capítulo 1 - Introdução

Hoje em dia é muito comum o desenvolvimento de aplicações em equipes que não seguem um padrão de desenvolvimento, e isso é grave devido ao fato de que torna-se difícil a compreensão de códigos quando um novo membro entra na equipe ou então quando há alterações no sistema que devem ser feitas.

Grandes empresas possuem um padrão de desenvolvimento que faz com que os desenvolvedores apliquem regras pré-determinadas e com isso dimunua um pouco o acoplamento entre as diversas partes no código do sistema, e isso torna o desenvolvimento melhor, pois aumentam a produtividade, diminuindo o tempo de mão-de-obra. Além disso, esses padrões são bons para que quando um novo membro ingressar na equipe, adapte-se facilmente e não perca tempo entendendo códigos de outras pessoas.

Um dos padrões mais utilizados hoje é o padrão MVC (model-view-controller), tal padrão separa a camada de interface com o usuário da camada de negócios. Existem dezenas de *frameworks* que auxiliam a fazer esse trabalho, cada um com seus pontos positivos e negativos, o único problema, é que quando falamos em desenvolvimento Java, surge a idéia de que Java é feito exclusivamente para *web*, e isso não é verdade. A grande maioria desses *frameworks* auxiliam nesse desenvolvimento para *Web*. Quando queremos trabalhar com aplicações *desktop*, temos que nos desdobrar para criar códigos “legíveis” para atualizações futuras. Pensando nisso foi elaborado uma ferramenta para auxiliar equipes nesse quesito, e além de tudo, posibilitar o seu uso em aplicações pequenas, por ser simples de configurar e simples de programar sob sua arquitetura.

O *gluemvc* está sob os termos da licensa LGPL (Lesser General Plublic License) da GNU (GNU's Not UNIX).

Esse tutorial foi escrito baseado na versão 0.1.x que está disponível para *download* no endereço http://sourceforge.net/project/showfiles.php?group_id=156866.

Capítulo 2 - Arquitetura

O *gluemvc* é um *framework* que encapsula alguns padrões bastante utilizados no desenvolvimento J2EE (apenas os conceitos são os mesmos, você pode e deve utilizá-los para o desenvolvimento sobre a plataforma *desktop*), esses padrões serão vistos posteriormente e também serão especificados a sua função dentro de uma aplicação.

Ainda falando sobre a arquitetura, vale lembrar que a aplicação de seus próprios *patterns* são bem-vindos, pois apenas utilizando o *gluemvc*, a interface entre a camada de negócios e a camada de interface com o usuário não é implementada, isso cabe a você fazer, seja implementando um *SessionFacade* ou então um *SessionBean* ou até mesmo um outro tipo de comunicação que você melhor se adapta. No **capítulo 5** será apresentado um exemplo contendo uma implementação da interface entre as duas camadas mencionada acima utilizando o *pattern SessionFacade*.

2.1 Patterns encontrados no framework

BusinessService – padrão que é utilizado para processar as informações de um dado objeto e distribuí-lo para o seu respectivo DAO (veja abaixo). Funciona mais ou menos dessa maneira: recebe um objeto passado pela interface de comunicação entre a interface com o usuário e a camada de negócios, aplica as regras de negócios e se tudo ocorrer como o previsto, esse objeto é passado para seu DAO, que o persiste. Serviços são os controladores da aplicação, podem ser criados implementando a *interface* **Service**, ou então extendendo a classe **AbstractService**. Algo muito importante, será comentado e exemplificado mais a frente, porém vou chamar a sua atenção aqui mesmo, é o fato de que para a arquitetura funcionar corretamente, seria de bom gosto que toda a vez que algum método das classes que você criar e que extenderem **AbstractService**, chamem o método correspondente da classe pai, pois caso o contrário, as regras de negócio (veja abaixo) não serão carregadas e processadas automaticamente.

BusinessRule – regras de negócio são as “etapas” que o seu caso de uso deve passar para que possa ser efetivado. Por exemplo, temos um caso de uso chamado “Sacar dinheiro”, para o usuário poder “sacar dinheiro”, é necessário que: a conta seja válida e exista saldo na conta. Issos são regras de negócio. Dentro do *framework* existe uma *interface* chamada **BusinessRule**, o programador pode optar por implementar essa interface ou então extender a classe **AbstractBusinessRule**, que já implementa essa *interface*. A seguir veremos mais detalhes sobre a implementação.

DAO (Data Access Object) – esse é um padrão bastante útil para qualquer tipo de aplicação. Ele persiste um dado. Praticamente toda a aplicação possui dados (objetos) que deverão ser salvos em arquivos, banco de dados ou até mesmo em memória, então cabem aos DAOs essa tarefa. Você pode criar os seus DAOs a partir da *interface* **DAO** ou então extendendo a classe **AbstractDAO**. Ambos possuem quatro métodos, que como todos nós sabemos são os básicos para o trabalho com JDBC ou outra maneira de persistência, *insert, update, delete* e *search*.

NOTA: Existe diversos outros padrões de projetos subentendidos dentro do *framework*, mas são estruturalmente desinteressantes, ou seja, eles são úteis apenas para o próprio framework, não são aplicáveis aos seus sistemas.

Capítulo 3 - Instalação

Por ser uma biblioteca de desenvolvimento, não é necessário a instalação do *gluemvc,* porém, para usá-lo é necessário adicioná-lo ao *classpath* de seu sistema ou da sua IDE de desenvolvimento.

3.1 Como adicioná-lo ao classpath

3.1.1 ECLIPSE

-Clique com o botão direito em cima de seu projeto;
-Selecione a opção *Properties*.
-Clique no ítem *Java Build Path* selecionando a aba *Libraries*.
-Clique no botão *Add JARs...* caso a gluemvc-x.x.x.jar esteja mapeado no diretório onde encontra-se o seu projeto, ou então clique no botão *Add External JARs...* e selecione na árvore de diretórios a localização da *lib*.

3.1.2 NETBEANS

3.1.3 CLASSPATH DO SISTEMA

-Compile sua aplicação usando
javac -classpath .:<localização do gluemvc-x.x.x.jar> <seu Main.java>
no Linux ou
javac -cp .;<localização do gluemvc-x.x.x.jar> <seu Main.java>
no Windows.
-E para rodar utilize:
java -classpath .:<localização do gluemvc-x.x.x.jar> <seu Main>
no Linux ou
java -cp .;<localização do gluemvc-x.x.x.jar> <seu Main>
no Windows.

Capítulo 4 - Configuração

A configuração do *gluemvc* é bastante simples. Você necessita criar um arquivo XML de configuração, esse arquivo deve conter os serviços (*services*), regras de negócio (*business rules*) e as classes de persistência (*DAOs*) especificados. Esse arquivo de configuração será chamado de *config.xml* e obrigatoriamente deverá ficar na pasta *conf* localizada logo abaixo do diretório raíz, ou seja, se você está desenvolvendo no diretório

/home/usuario/projetos/app
seu arquivo de configuração será encontrado em
/home/usuario/projetos/app/conf/config.xml

Segue abaixo um modelo do arquivo com a descrição detalhada do mesmo:

```
<?xml version="1.0" encoding="UTF-8"?>

<glue description="Arquivo de configuracao para gluemvc" version="1.0.0">

	<daos description="DAOs da aplicacao">
		<dao name="mydao" path="app.MyDAO"/>
	</daos>

	<services description="Services da aplicacao">
		<service daoref="mydao" name="mys1" path="app.MyService"/>
		<service daoref="" name="mys2" path="app.MyService2"/>
	</services>

	<busrules description="Regras de negocios da aplicacao">
		<br service="mys" description="" name="" path="app.ValidaRN"/>
		<br service="mys" description="" name="" path="app.DebitaRN"/>
		<br service="mys2" description="" name="" path="app.ProcessaRN"/>
	</busrules>

</glue>
```

4.1 Construindo o config.xml

Como é possível notar, o arquivo de configuração é dividido em três seções principais, *daos*, *services* e *busrules*. Todas elas giram em torno de suas classes de serviço, pois é a partir dele que serão invocadas as regras de negócio e também a classe que fará a persistência do mesmo.

Atributos do arquivo XML:

Services são criados com as seguintes propriedades:
-*name*: nome do serviço, identificador do mesmo.
-*path*: localização, em sua arquitetura, em que o seu serviço se encontra.
-*daoref*: informa o *DAO* referente àquele serviço. (pode ser nulo)

BusinessRules são criadas com as seguintes propriedades:
-*name*: nome da regra, identificador do mesmo. (pode ser nulo)
-*description*: breve descrição sobre o que essa regra faz. (pode ser nulo)
-*path*: localização, em sua arquitetura, em que a sua regra de negócio se encontra.
-*service*: identificador do *service* que a regra pertence.

DAOs são criadas com as seguintes propriedades:

-*name*: nome do DAO, identificador do mesmo.

-*path*: localização, em sua arquitetura, em que o seu DAO se encontra.

Capítulo 5 - Implementação

A implementação de um programa usando o *gluemvc* deve ser uma tarefa simples (ao menos do ponto de vista de acoplamento entre as duas plataformas).

A seguir será apresentado alguns exemplos de código, exemplificando o funcionamento do componente.

5.1 Exemplos

Exemplo 1: Inserindo um cliente no banco de dados

```
/* imports */
public class MyService extends AbstractService {
    public void insert(Object o) throws ServiceException {
      super.insert(o);  // IMPORTANTE!!!
    }
}
```

Classe do tipo *Service* que apenas recebe um objeto para tratamento.

As classes *services*, como já foi explicado anteriormente, são os controladores da sua aplicação. O exemplo acima encontra-se com pouco código devido ao fato de a chamada às regras de negócio e também à classe de persistência está sendo feita na classe "pai" e as referência encontram-se no arquivo de configuração *config.xml*. Isso é o que facilita a intergração e ao mesmo tempo o desacoplamento entre as diferentes camadas encontradas nesse exemplo.

NOTA: No código fonte acima, o termo "IMPORTANTE" refere-se ao fato de sempre que você extender a classe *AbstractService* e implementar algum de seus métodos, deverá, obrigatoriamente, chamar o método correspondente da classe pai, pois caso o contrário as regras de negócio não serão executadas e o objeto não será persistido. Se você fizer uso de um método semelhante ao descrito acima, ou seja, um método vazio onde não há código, não sobreescreva o método, como foi feito nessa caso, apenas deixe tudo por conta da super classe.

```
/* imports */
public class BusinessRuleOne extends AbstractBusinessRule {

     public boolean process(Object o) {
          Client c = (Client) o;

          if (c.getName()==null && c.getName().equals(""))
               return false;

          if (c.getYearsOld() < 0) return false;

          if (c.getYearsOld() < 18) {
               c.setDrunk(false);
          } else {
               c.setDrunk(true);
          }

          return true;
     }

}
```

Classe do tipo *BusinessRule* responsável pela validação do objeto.

Aqui é verificado se o nome do cliente não é nulo ou vazio, se não for, “ok”, pode continuar a transação, caso o contrário, o *false* já é retornado para que o desenvolvedor faça o tratamento adequado do erro encontrado.

Podem haver mais de uma regra de negócio para cada serviço, e também pode ser usado uma regra de negócio para vários serviços distintos.

Regras de negócio possuem três métodos importantes, *preProcess*, *process* e *postProcess*. O *process* é o único que o programador fica obrigado a implementar, ele é a regra de negócio propriamente dita. O *preProcess* e o *postProcess* são executados antes e depois de *process*, respectivamente, para que um objeto seja preparado ou alguma configuração carregada, caso necessário.

NOTA: O objeto *Client* mencionado nos exemplos não possui código para conferência nesse tutorial. Ele é uma entidade, um *Value Object*. Apenas conhecendo seus *GETS* e *SETS*, que aparecem nos outros códigos, é suficiente para entender o funcionamento do *framework*.

```
/* imports */
public class MyDAO extends AbstractDAO {

    private Connection getConnection() {
      Connection connection = null;
      Class driverClass = null;
         try {
             if (driverClass == null)
                 driverClass = Class.forName(
                                  "org.hsqldb.jdbcDriver");
         } catch (ClassNotFoundException e) {
             return null;
         }
         if (connection == null) {
             try {
                 connection = DriverManager.getConnection(
                              "jdbc:hsqldb:appdb","sa","");
                 connection.setAutoCommit(true);
             } catch (Exception e1) {
                 System.out.println(e1.getMessage());
             }
         }
         return connection;
    }

     public void insert(Object o) throws DAOException {
          Client c = (Client) o;

          String query = "insert into client values (" +
                         c.getId() + ", '" +
                         c.getName() + "', " +
                         c.getYearsOld() + ", '" +
                         c.isDrunk() + "')";

          try {
               Connection connection = getConnection();
               Statement st = connection.createStatement();
               st.execute(query);
          } catch (SQLException e) {}
     }
}
```

Classe de persistência, recebe o objeto e salva-o na base de dados.

Isso é apenas um exemplo de como o objeto pode ser persistido. Nesse caso ele está salvando-o em uma base de dados via JDBC (Java Database Connectivity), mas poderia estar sendo guardado em arquivo *TXT* ou *XML* ou até mesmo em memória, através de uma classe auxiliar que contém uma coleção de objetos, isso é possível e bastante comum.

Agora que já temos as classes que serão usadas para esse caso de uso e que são extendidas do *framework*, está na hora de conhecer as classes que não implementarão nenhuma *interface* do *framework*.

```
/* imports */
public class MainFrame extends JFrame implemets
ActionListener {
    /* atributios da classe */
    private JButton btSaveClient;

    public MainFrame() {
        /*  ...  */
    }

    /* métodos para iniciar os componentes e desenhar a
interface */

    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == btSaveClient) {
            SessionFacade sf = new SessionFacadeImpl();
            Client c = new Client();
            c.setId(...);
            c.setName(...);
            c.setYearsOld(...);
            sf.insertClient(c);
        }
    }
}
```

Esse código é responsável por desenhar um formulário na tela, para o usuário preencher, isso fica a critério de cada programador, o que importa nesse exemplo é mostrar como funciona a interface de comunicação entre esse *view* e a classe *MyService*.

```
public interface SessionFacade {

     public void insertClient(Client client);
    /* outros métodos
     refentes aos casos de uso */

}
```

A *interface SessionFacade* especifica TODOS os casos de uso de sua aplicação, é por aqui que os *views* acessarão a camada de negócio. Utilizando essa *interface* não é necessário alterar absolutamente nada caso a camada visual seja alterada, apenas implementar uma outra classe que implemente *SessionFacade*.

```
public interface SessionFacadeImpl implements SessionFacade
{

     public void insertClient(Client client) {
          Service service = XmlServiceFactory("client");
          service.insert(client);
     }

    /* implementação dos outros métodos */

}
```

Aqui está um exemplo de uma classe implementando a *SessionFacade*. Perceba que não há uma dependência direta entre essa interface e os serviços, tudo é feito através de um *Factory* (detalhes abaixo), sendo assim, caso haja a necessidade de trocar a camada de negócios, basta editar o arquivo de configuração, não necessitando alterar os códigos fontes. Isso também é possível caso seja desejável que a camada de modelo seja totalemnte modificada, substituindo os atuais DAOs usando JDBC, por DAOs usando arquivos XML.

5.2 Factories, como funcionam

Os f*actories* são utilizados para produzir um determinado tipo de classe. Isso facilita a manutenção e auxilia no desacoplamento da arquitetura da aplicação. No *gluemvc* podem ser encontradas as *interfaces ServiceFactory*, *BusinessRuleFactory* e *DAOFactory* e ainda as implementações dessas *interfaces*: *XmlServiceFactory*, *XmlBusinessRuleFactory* e *XmlDAOFactory*, respectivamente.

A princípio você não necessita utilizar os *factories* de DAOs e de BusinessRules, apenas os de serviço devem ser compreendidos. Como foi visto no ***Exemplo 1*** do **capítulo 5**, o *factory* de serviço será utilizado sempre dentro de uma implementação de *SessionFacade*, pois é essa a classe que recebe uma dada requisição da interface com o usuário, identifica qual o *Service* que corresponde ao seu devido tratamento, instancia-o, utilizado um *factory*, e invoca o método necessário.

Essas implementações trabalham com o arquivo de configuração *config.xml*, buscando nele o caminho completo de uma classe baseado em uma *String* passada por parâmetro. O que importa nesse caso é que nem todo mundo admira trabalhar com arquivos XML ou então prefere guardar dados referentes a sua implementação em algum lugar mais seguro, como uma base de dados ou algum arquivo criptografado, impossibilitando o usuário do sistema de alterar esse dados. Sendo assim é possível implementar as *interfaces* mencionadas anteriormente e então, carregar as classes da maneira que você desejar.

Por padrão, a classe *AbstractService* utiliza o *factory XmlBusinessRuleFactory* para carregar as regras de negócio e o *factory XmlDAOFactory* para carregar a classe responsável pela persistência de objetos, DAO, correspondente, caso você deseje ler as propriedades dessas classes de um local ou arquivo diferente do arquivo de configuração padrão, ao criar alguma classe que extenda *AbstractService*, utilize o método *setBusinessRuleFactory* e *setDAOFactory* para informar o *BusinessRuleFactory* e o *DAOFactory*, respectivamente, criados por você.

Um exemplo:

```
/* imports */
public class MyService extends AbstractService {

    public MyService() {
        setBusinessRuleFactory(
             new MyBusinessRuleFactory());
        setDAOFactory(new MyDAOFactory());
    }

    public void insert(Object o) throws
    ServiceException {
      /* implementação do método */
    }
}
```

Pronto, assim quando a classe for instanciada, os seus *factories* já estarão incorporados à ela.

A documentação completa e detalhada de todas as classes e funcionalidade do *framework* pode ser encontrada na *home page* do projeto.

5.3 Conditions

Condition é uma espécie de propriedade de condição, já *Conditions* representa um repositório de *Condition*. Com *Condition* você pode definir uma propriedade que poderá ser recuperada posteriormente (dentro de métodos *search* de classes DAO), essas condições serão armazenadas dentro de uma estrutuda de dados chamada de *Conditions*, permitindo assim enviar o número de condições que você desejar para construir as suas pesquisas.

Para criar uma *Condition*, faça assim:

```
...
Service s = XmlServiceFactory.getService("myService");
Condition c1 = new Condition("12/12/2006");
Conditions conds = new Conditions();
conds.addCondition("firstDate", c1);
// ou faça assim
conds.addCondition("finishDate", "14/12/2007");
s.search(conds);
```

Para resgatar uma *Condition*, utilize o exemplo abaixo:

```
...
Condition c1 = conds.get("firstDate");
String firstDate = c1.getCondition();

...
if (anotherDate.equals(fisrtDate)) {
    List l = new List();
    l.add(anotherDate);
    return l;
}
...
return null;
```

A classe *Condition* "guarda" uma *String* como propriedade, mas já foi constatado que não é muito bom e nem eficiente. No código acima, por exemplo, em um caso real trabalharíamos com objetos do tipo *Date* e isso nos obrigaria a fazer diversas conversões de tipos para a nossa lógica funcionar corretamente. Já está programado para a versão 0.2.x, a troca da arquitetura *Condition*, visando não mais armazenar as propriedades como *String*, mas sim como *Object*. Com isso seria possível resgatar o próprio objeto a ser utilizado como condição, e não uma pseudo-condição.

Consulte o **capítulo 6** para saber sobre as demais mudanças previstas.

Capítulo 6 – TODO

Nas novas *releases* que serão lançadas do *gluemvc*, espera-se que a produção de regras de negócio seja feita de maneira mais dinâmica, utilizando coleções de regras, ao invés de várias classes que representam regras de negócio.

Outra *feature* que está sendo programada é a de criar uma classe a partir de *AbstractDAO* que já contenha uma conexão com JDBC formada, bastando ao desenvolvedor apenas informar o *driver* de conexão e a *URL* do banco. Com isso a conexão não precisa ser criada a cada método que persistirá um objeto e também não será necessária a criação de uma classe auxiliar para essa tarefa.

Modificar a maneira como é montada uma *Condition*, substituíndo o atual atributo do tipo *String* por um do tipo *Object*. Espera-se que com isso a necessidade de conversões de tipos diminua drásticamente.

Capítulo 7 - Colaboração

7.1 Ferramentas

Esse projeto foi desenvolvido utilizando apenas ferramentas livres, segue a lista das ferramentas utilizadas e seus endereços na web:

IDE de desenvolvimento
Eclipse – http://eclipse.org

Ferramenta de build
Apache Ant – http://ant.apache.org

Framework de teste
Junit – http://junit.org

7.2 Como contribuir com o projeto

Se você tem interesse em colaborar com o projeto *gluemvc*, entre em contato com o desenvolvedor pelo e-mail giulianobg@users.sourceforge.net.

Atualmente, as necessidades do projeto são:
-Aplicações de exemplo utilizando o *framework* desenvolvido;
-Tradutores para esse manual;
-Desenvolvedores de páginas *web* para a produção da *home page* do projeto.

Se tiver alguma idéia de *feature* para expor, entre em contato também.