Uma Plataforma de Software para o Estudo Interativo de Métodos e Algoritmos Econométricos: Mecanismos de Arquitetura

Draft Inicial, versão 2 (Março/2008) – Carlos Duarte do Nascimento

Introdução Este documento tem por objetivo detalhar as tecnologias (linguagens, ferramentas e frameworks) e metodologias (padrões de projeto) escolhidas para a aplicação que irá implementar a plataforma proposta, expondo as motivações por trás de cada escolha.

As escolhas definidas aqui foram norteadas por dois pré requisitos: que todas as metodologias tenham referencial acadêmico e implementações bem-sucedidas; e que todas as ferramentas utilizadas sejam gratuitas e de código livre, garantindo que a aplicação possa ser ampliada e modificada por qualquer entidade interessada, sem que hajam restrições de qualquer natureza.

Linguagem de Programação É necessário fazer duas escolhas neste tópico: a da linguagem a ser utilizada para construir o aplicativo, e a da linguagem através da qual os algoritmos serão descritos para a mesma.

Para o aplicativo, é preciso ter uma linguagem que ofereça performance, escalabilidade, suporte ao desenvolvimento para a web e facilidade para trabalhar com bancos de dados relacionais (ferramenta indispensável para o volume de dados gerado pelo uso em larga escala, vide adiante). Também é importante que a linguagem não seja excessivamente obscura ou limitada a um nicho de mercado, já que um dos objetivos é propor uma plataforma que possa ser ampliada e melhorada por quem tenha interesse.

Já o algoritmo exige uma linguagem que possa ser compilada ou interpretada dinamicamente, que tenha respaldo acadêmico (para aproveitar o código e, mais importante, o conhecimento pré-existente no corpo docente) e, preferencialmente, que já possua um ambiente de compilação/runtime livre (a implementação de tal ambiente não as complexidades inerentes a este tipo de implementação fogem ao escopo deste trabalho).

Apesar das naturezas relativamente divergentes das demandas mencionadas acima, os avanços recentes em linguagens de programação [citação] oferecem linguagens e ambientes flexíveis o suficiente para atender a ambas as demandas – e o uso de uma linguagem única simplifica o projeto e reduz a barreira de entrada para novos desenvolvedores.

Algumas das linguagens "clássicas" consideradas foram:

C/C++: A melhor opção em termos de performance. Por ter acesso aos mais diferentes tipos de bilbiotecas nos diversos sistemas operacionais existentes, também não apresentaria problemas para trabalhar com banco de dados ou com web. No entanto, a programação nesta dupla é bastante sujeita a erros, e muitas vezes é preciso escolher entre a flexibilidade do C++ e a performance do C. Além disso, é necessário um esforço extra para garantir a compabilidade entre diferentes plataformas. Tais fatores elevariam o tempo do projeto e colocariam uma barreira à entrada de novos desenvolvedores – isso sem falar que dificilmente a programação dos algoritmos seria feita de forma didaticamente viável nela.

Pascal: Tem a seu favor um excelente balanço entre performance e facilidade de programação (por ser mais fortemente tipada do que C e executar muitas das verificações de erros comuns em tempo de execução), além de possuir extensões de orientação a objeto e implementações livres (como o Free Pascal). No entanto as universidades já não tem mais incluído esta linguagem em seus currículos (Java é o substituto mais comum), o que se reflete em reduzida disponibilidade de programadores.

FORTRAN: Um dos pontos fortes é a vasta quantidade de algoritmos econométricos disponíveis na literatura já codificados nesta linguagem. No entanto, a linguagem oferece poucas

facilidades para a programação na web, e a mão-de-obra disponível é muito limitada ao meio científico/acadêmico.

O equilíbrio entre as demandas funcionais e não-funcionais levou à escolha da linguagem **Java** para a primeira implementação. Caso esta escolha não seja a mais apropriada no futuro, recomenda-se o uso de linguagens dinâmicas como **Ruby**, **Python** ou **LISP**, ou, alternativamente, como linguagens estáticas com suporte a reflexão, como **C#** ou **Objective-C** – observando-se o mapeamento dos mecanismos de arquitetura para recursos equivalentes em cada uma delas.

A versão 6 da plataforma Java permite que o compilador seja invocado através de APIs de alto nível. Isto permitirá a execução interativa dos algoritmos econométricos de forma independente de sistema operacional ou plataforma.

Ambiente de Desenvolvimento, Compilação e Publicação É importante que o processo de compilação e publicação (*deploy*) da aplicação sejam completamente automatizados, de forma que um novo desenvolvedor possa facilmente descarregar o código existente de um repositório, testá-lo e implementar novas características.

A grande quantidade de frameworks envolvidos no processo gera uma dificuldade adicional: o gerenciamento de dependências. Tento isto em vista, usaremos o **Maven** não apenas para executar a compilação e publicação, mas também para recuperar automaticamente todas as bibliotecas (.JAR) de frameworks utilizados a partir da Internet.

O código-fonte não irá exigir nenhum IDE (ambiente de desenvolvimento) em particular. No entanto, o **Eclipse** será utilizado como ferramenta base, sempre tomando o cuidado de não tornar a ferrametna dependente dele.

Testes Automatizados É muito importante que o sistema mantenha os resultados consistentes, mesmo com a implementação de novas funcionalidades. Para tanto, a criação de testes automáticos (unitários e funcionais) durante o processo de desenvolvimento (e não como um detalhe adicional) é indispensável [citação de test-driven development].

O framework **JUnit** é uma escolha natural para a implementação de testes automatizados, devido à sua natureza não-intrusiva e ao uso difundido em projetos Java. Mesmo testes funcionais automáticos podem ser feitos através dele (havendo a possibilidade de expandi-lo com o uso da extensão **JFunc**).

Arquitetura Sob a perspectiva da arquitetura geral, o sistema pode ser visto como uma coleção interativa de CRUDs[1] (as exceções ficam por conta da execução interativa de algoritmos e da importação de dados), sendo, portanto, razoável trabalhar com a tradicional [citação] separação em três camadas:

1. uma camada de operações (back-end) tais como: armazenamento de dados, execução de algoritmos e conversão de formatos, cuja implementação é detalhada adiante;

2. uma camada intermediária de "fachada" [citação: Session Facade Pattern], agrupando as operações de alto nível. Esta camada será implementada através de classes e métodos estáticos simples [citação: POJO];

3. uma camada de interface (front-end), utilizando MVC e outros princípios detalhados a seguir.

[1] Create-Read-Update-Delete, acrônimo para módulos que efetuam estas quatro operações básicas sobre algum tipo de entidade.

As melhores práticas de desenvolvimento de software muitas vezes demandam a implementação de diversos padrões de projeto [citação]. Felizmente, a plataforma Java conta com diversos frameworks que implementam tais padrões de projeto, economizando esforço e tornando o código mais enxuto e focado no problema educacional.

O restante desta sessão é dedicado a descrever e justificar algumas destas práticas, definindo (quando aplicável) os frameworks selecionados para a implementação das mesmas.

Mapeamento Objeto-Relacional

Ao longo das últimas décadas, os sistemas gerenciadores de banco de dados relacionais (RDBMS) simplificaram o armazenamento de dados através da introdução da abordagem relacional de representação dos mesmos, implementada em pacotes de software de baixo custo, de forma isolada da aplicação principal, permitindo ao desenvolvedor concentrar-se no domínio específico do problema computacional a ser resolvido.

Além disso, a popularização dos RDBMS permitiu o uso de recursos computacionais relativamente limitados (tais como microcomputadores) para a execução de tarefas de manipulação de dados anteriormente restritas a sistemas de grande porte (e elevado custo de operação e manutenção), razão pela qual o uso de um RDBMS é indicado em qualquer sistema no qual a manipulação indireta dos dados não represente impacto na performance.

Tal característica, aliada à importância que os dados representam para as organizações (chegando, em muitos casos, a ser mais valiosos que os aplicativos ou os meios físicos nos quais eles são armazenados e processados), fez com que muitos dos sistemas projetados neste período tivessem o modelo relacional de banco de dados como base do seu projeto – o *software* que manipularia estes dados era pensado de forma secundária, quase que conseqüência direta do desenho do banco.

Com a introdução das técnicas de desenvolvimento de software orientado a objeto – outro artefato que aumentou o nível de abstração com o qual os projetistas de software lidam com o domínio dos problemas (e, portanto, a sua produtividade) – surgiu uma nova abordagem: proponentes destas técnicas defendem que o sistema deve ser modelado sob o ponto de vista de suas classes, tratando o banco de dados como um mero armazém de objetos.

A plataforma proposta neste trabalho segue esta nova abordagem, para a qual se coloca um problema: como representar a riqueza gramatical dos elementos da orientação a objeto (tais como herança, polimorfismo e navegabilidade) dentro do sistema de modelagem relacional dos RDBMS? Esta questão tem dois matizes: a metodologia para mapear estas características e a forma de implementá-la (evitando a redundância de código).

Este problema não é novo [citação], tampouco exclusivo desta aplicação. A técnica para resolve-lo é denominada mapeamento objeto-relacional, e na plataforma Java existem diversos framework que a implementam [citar exemplos].

Dentre eles, optamos pelo **Hibernate**. Trata-se de um framework de código totalmente livre, bastante flexível e amplamente utilizado pela comunidade Java. Além disso, a versão 3 permite o uso de *annotations*, isto é, do desenho do mapeamento sobre o próprio código. Esta característica torna os arquivos de mapeamento (que usualmente demandam muito tempo e acrescentam um passo extra na compreensão do código) dispensáveis, o que, por si só, já justifica o seu uso.

Apresentação e MVC (Model / View / Controller)

Devido à sua natureza de interação com o usuário, a camada de apresentação é uma das mais sujeitas a alterações. Além disso, seu fluxo pode se tornar bastante complexo, o que favorece a duplicação de código desnecessária.

O padrão de projeto Model/View/Controller (MVC) tem se demonstrado útil na redução destes problemas. Nele, a camada de apresentação é segregada em dois tipos de componentes:

view (composta pelas diversas interfaces[2] do sistema, e desprovida de qualquer código que não esteja relacionado à interação com o usuário e à pré-validação dos dados introduzidos por ele) e *model* (código que responde a ações imperativas do usuário, tais como submeter um formulário de dados ou solicitar uma funcionalidade).

View e Model operam de forma totalmente independente: componentes de model respondem às solicitações utilizando as camadas inferiores e retornando algum tipo de status (ex.: "sucesso", "operação inválida", etc.), e componentes de view apresentam dados anexados a eles e retornam os dados novos ou alterações feitas pelos usuários.

A conexão entre eles é feita pelo controller: um componente que, para cada solicitação da view, dispara um ou mais componentes do model, e, conforme o resultado, apresenta uma nova view. Todo o fluxo é mantido neste componente (no código ou em um arquivo de configuração), desacoplando o código e oferecendo uma visão de alto nível que torna fácil identificar componentes reutilizáveis e/ou o impacto de quaisquer mudanças [Citação MVC].

Embora seja possível adotar a filosofia MVC através do desenvolvimento direto, muito trabalho pode ser popupado através do uso de um framework MVC que, dentre outras coisas, implemente um controller configurável e auxilie na passagem de dados entre model e view (tarefa que se torna complexa à medida em que se considera a generalidade do HTML no tocante a formulários de dados, e o desejo de usar técnicas como AJAX para aumentar a usabilidade da aplicação). De início, foram considerados os frameworks mais utilizados atualmente, a saber:

Struts: Um dos frameworks mais tradicionais em Java, tem como vantagens uma biblioteca de apresentação bastante rica e evolução constante. Sua maior limitação é a grande quantidade de código/configuração necessários para definir o fluxo da aplicação;

Spring MVC: É parte do framework Spring, o que facilitaria a sua integração. Mas também sofre do mal de exigir muita configuração, sem apresentar maiores atrativos que compensem o fato;

Tendo em vista que as alternativas padrão não atendem às necessidades do projeto, foram considerados frameworks com menor base de usuários, tais como **VRaptor**, Jaffa e **Stripes**. Este último, por contar com excelente documentação e muitos exemplos na web, foi escolhido para a implementação.

Finalmente, é importante salientar que existe uma outra alternativa para otimizar o desenvolvimento da camada de apresentação: o uso de arquiteturas baseadas em componentes, tais como **Wicket**, **WebWorks** e **JSF** (os dois primeiros são frameworks, o último é uma especificação para este tipo de arquitetura, definida sob supervisão da empresa que desenvolve o Java e seguida por diversos frameworks), sobretudo pela agilidade que oferecem na prototipação e na criação de interfaces. No entanto, aplicações desenvolvidas sob este tipo de arquitetura não possuem a "tolerância a mudanças" que o MVC proporciona (e estas mudanças seguramente ocorrerão à medida em que a plataforma for expandida), razão pela qual descartamos tal tipo de solução.

Inversão de Controle / Injeção de Dependências

Tendo em perspectiva que a aplicação resultante deste projeto será uma base para o desenvolvimento de outros sistemas, é importante que a mesma seja de fácil compreensão e manutenção.

Um dos grandes obstáculos para a manutenção de projetos de software é o acoplamento excessivo entre os seus diferentes módulos e camadas. Ainda que se use (e usamos) boas práticas de separação das mesmas (tais como a arqutetura Model-View-Controller e o modelo de três camadas), se estas camadas apresentarem excessiva dependência cruzada, pequenas alterações irão demandar grandes esforços de recodificação e teste.

A inversão de controle [citar: Martin Fowler] (IoC) é uma técnica de projeto que aborda o

[2]Por interfaces aqui entendemos as unidades da aplicação com os quais o usuário lida. Exemplos incluem "telas" de mainframe, caixas de diálogo em aplicações GUI e (no nosso caso) páginas web.

problema do acoplamento subvertendo a maneira tradicional com que um módulo do sistema solicita funcionalidade a outro módulo (daí o nome). O princípio fundamental é que um módulo que dependa de outro para executar a sua funcionalidade não o chama explicitamente – ao invés disso ele manifesta esta dependência de alguma forma, e o ambiente operacional cuida de oferecer o componente que melhor ofereça o tipo de serviço necessário.

Isso faz com que o módulo se concentre na sua própria funcionalidade – ao invés de misturar este código com o código que cuidará da iteração com a dependência. Há um ligeiro aumento na quantidade de código devido à necessidade de formalizar a dependência através de suas características (e não através da chamada direta do módulo que satisfaz a dependência) – o exemplo canônico de Fowler mostra essa diferença. Mas isto é largamente compensado pelo desacoplamento obtido, e a clareza do código não é prejudicada.

Existem várias formas de implementar o princípio de IoC, sendo que a Injeção de Dependências é bastante popular por reduzir a quantidade de código envolvida no processo. Nela, o módulo que oferece a funcionalidade apresenta uma interface não apenas para as tradicionais chamadas, mas também para as dependências delas (ex.: conexões de banco de dados, canais de saída, etc.). O módulo que solicita a funcionalidade o faz como na programação tradicional (chamando o método exposto), mas um framework media estas chamadas e fornece as dependências necessárias de forma apropriada

Para trabalhar sob esta abordagem utilizamos o Spring – um framework que tornou-se referência em aplicações Java. Uma de suas vantagens é se integrar de forma automatizada com outros frameworks (notadamente Hibernate), reduzindo ainda mais a quantidade de código na nossa aplicação. O uso será bastante comedido (evitando o anti-padrão que duplica cada classe do sistema em uma interface [citação]), limitando-se aos pontos em que uma eventual substituição de componentes se mostre como uma possibilidade interessante.