Reengenharia de um Simulador de Dinâmica de Sistemas com Aplicação de Padrões de Projeto

Brian Mazini Siervi

JUIZ DE FORA

ABRIL, 2013

Reengenharia de um Simulador de Dinâmica de Sistemas com Aplicação de Padrões de Projeto

Brian Mazini Siervi

Universidade Federal de Juiz de Fora

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Bacharelado em Ciência da Computação

Orientador: Prof. Ciro de Barros Barbosa

JUIZ DE FORA

ABRIL, 2013

Reengenharia de um Simulador de Dinâmica de Sistemas com Aplicação de Padrões de Projeto

Brian Mazini Siervi

MONOGRAFIA SUBMETIDA AO CORPO DOCENTE DO INSTITUTO DE CIÊNCIAS EXATAS DA UNIVERSIDADE FEDERAL DE JUIZ DE FORA, COMO PARTE INTEGRANTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE BACHAREL EM CIÊNCIA DA COMPUTAÇÃO.

Aprovada por:

Prof. Ciro de Barros Barbosa
Doutor em Ciência da Computação, Twente, 2001.

Prof. Jairo Francisco de Souza
Doutor em Informática, PUC-Rio, 2012.

Prof. Michel Heluey Fortuna
Doutor em Engenharia de Sistemas e Computação, UFRJ, 2009.

JUIZ DE FORA

04 DE ABRIL, 2013

Dedico este trabalho, a ti, minha esposa, pela paciência, amor, compreensão e dedicação que sempre teve a mim. Reservo a ti, a gratidão eterna, de quem ama com todo o coração e a admira com toda consciência de um ser.

Resumo

Após 2 (dois) anos de desenvolvimento, com a participação de 1 (um) coordenador e 3 (três) colaboradores, a estrutura do Web-Based System Dynamics Simulator (WSDS) apresentou sinais de que não poderia seguir em frente, devido à falta de boas práticas de programação de *software*, o que inviabilizava a sua manutenção. No contexto dessa ferramenta de simulação *web*, o presente trabalho aborda a implementação de melhorias essenciais a ela, de forma que o acesso simultâneo de usuários seja melhor aproveitado, uma maior variedade de métodos diferenciais seja aplicada e a manutenibilidade da ferramenta seja garantida com a utilização de padrões de projetos, realizando-se portanto, a sua reengenharia. Criada na década de 60, como sendo uma subcategoria da Teoria de Sistemas, a Dinâmica de Sistemas (DS) ganhou destaque em uma matéria lecionada no Massachusetts Institute of Technology (MIT). A facilidade de visualização de complexos sistemas através da abordagem gráfica da DS despertou muito interesse na época, e após algumas décadas, começaram a surgir *softwares* que uniram a esta abordagem o poder computacional. Até então, tais *softwares* permitiam a modelagem através de uma linguagem gráfica, enfatizando o caráter intuitivo da modelagem visual. Para solucionar essa limitação, em 2001 foi criada uma linguagem textual para modelagem, juntamente com ferramentas de simulação que demandavam um processo de compilação do modelo para obtenção de um código executável. Em uma diferente vertente, surgiu no ano de 2011, o WSDS, que a partir dessa linguagem textual gera, de forma interpretada, a simulação computacional, possibilitando uma interação do usuário com o processo de simulação.

Palavras-chave: Teoria de Sistemas, Dinâmica de Sistemas, Simulação Computacional, Padrões de Projetos.

Abstract

After 2 (two) years of development, with the participation of 1 (one) coordinator and 3 (three) employees, the structure of the Web-based System Dynamics Simulator (WSDS) showed signs that he could not move forward due to lack best practices in software development, making it impossible to maintain it. In the context of this simulation tool web, this paper discusses the implementation of key improvements to it, so that simultaneous access of users to be better used, a greater variety of differential methods is applied and maintainability of the tool is guaranteed with the use of design standards, performing therefore its reengineering. Built in the 60s, as a subcategory of Systems Theory, System Dynamics (SD) gained prominence in a matter taught at the Massachusetts Institute of Technology (MIT). The ease of visualization of complex systems through graphical approach of DS sparked much interest at the time, and after a few decades, began to emerge software that joined this approach the computational power. Until then, these softwares allow modeling through a graphical language, emphasizing the character's intuitive visual modeling. To solve this limitation, in 2001 created a textual language for modeling along with simulation tools that required a build process template to obtain an executable code. In a different aspect, appeared in 2011, the WSDS that from that generates textual language, so interpreted, the computational simulation, enabling user interaction with the simulation process.

Keywords: System Theory, System Dynamics, Computer Simulation, Design Patterns.

Agradecimentos

A Deus, por me guiar e abençoar.

Aos meus pais, pelo amparo aos meus estudos e lapidação de minha índole.

À minha irmã, pela força nos momentos mais difíceis.

À minha sogra, que fez o impossível para ver o meu sucesso e felicidade.

Ao meu sogro, pela palavra amiga.

Em especial, ao professor e orientador, Ciro de Barros Barbosa, pela ajuda que nunca foi negada e comprometimento com este trabalho.

E a todos amigos de classe e professores do Departamento de Ciência da Computação da Universidade Federal de Juiz de Fora, que me ajudaram nesta conquista.

"Penso noventa e nove vezes e nada descubro; deixo de pensar, mergulho em profundo silêncio - e eis que a verdade se me revela".

Albert Einstein

Sumário

Lista de Figuras

Lista de Abreviações

DCC	Departamento de Ciência da Computação
UFJF	Universidade Federal de Juiz de Fora
DS	Dinâmica de Sistemas
SC	Simulação Computacional
WSDS	Web-Based System Dynamics Simulator
ES	Engenharia de Software
RS	Reengenharia de Software
ERS	Engenharia Reversa de Software
EPS	Engenharia Progressiva de Software
SEDOs	Sistemas de Equações Diferenciais Ordinárias
LST	Linguagem de Simulação Textual
AST	Árvore de Sintaxe Abstrata
TI	Tecnologia da Informação

1 Introdução

1.1 Contexto, Motivação e Objetivos

No meio computacional quando um software está desatualizado, obsoleto, e possui uma fraca documentação (ou nem mesmo a possui), muito se fala em Engenharia Reversa de Software (ERS), que nada mais é do que um componente da Engenharia de Software[1] (ES). Componente este que tem o objetivo de iniciar uma análise de um baixo nível de abstração, geralmente representado pelo código-fonte do software, até um nível mais elevado, onde é finalizada ao se obter os requisitos intrínsecos do software em questão. Este fluxo, também é conhecido como sendo o reverso da Engenharia Progressiva de Software[2] (EPS) (CHIKOFSKY e CROSS, 1990).

Outro importante componente da ES, que muitas vezes faz uso da ERS, é a Reengenharia de Software (RS). A RS é descrita por alguns autores, podendo-se citar Warden e Pressman, como sendo um incremento relevante de funções que não alteram a atividade fim de um software (WARDEN, 1992; PRESSMAN, 1995). Outros, como Sommerville, definem a RS como sendo uma modificação parcial ou total em um determinado software, a fim de realizar a melhoria de sua manutenibilidade (SOMMERVILLE, 2001).

Embora os conceitos da ERS e a RS de software sejam de fácil interpretação, a sua aplicabilidade exige um estudo embasado onde se determine até que nível de detalhamento e/ou abstração deve-se atingir. Um fator relevante a essa aplicabilidade é o custo/benefício de se realizar a Reengenharia em um software, visto que nos casos ideais, o custo dessa atividade tende a ser menor do que o custo de produção inicial do software.

Neste trabalho aplica-se a RS à ferramenta WSDS, que possui um porte médio, utiliza diferentes linguagens de programação, pode ser executada em diferentes sistemas operacionais e foi desenvolvida por diferentes desenvolvedores que não mantiveram con-

[1]Engenharia de Software: termo criado por Friedrich Ludwig Bauer, definido como sendo: "o estabelecimento e o uso de sólidos princípios de engenharia para que se possa obter economicamente, um software que seja confiável e que funcione eficientemente em máquinas reais."

[2]Engenharia Progressiva de Software: Reflete um clico de vida natural do desenvolvimento de *software*, partindo de um nível mais elevado de abstração até o mais baixo nível.

tato entre si, e que geraram portanto, um código-fonte de difícil manutenção.

No caso de RS aplicado a ela, não houve outro custo senão o tempo despendido. E o principal benefício, foi a continuidade de utilização dessa ferramenta, agregando melhorias que a tornam mais manutenível. Sendo este trabalho apenas uma prova de conceito relativa à área de RS, não são enfatizadas nele as técnicas propriamente ditas da ES. Por outro lado, são enfatizados alguns conceitos da RS que devem ser aplicados à ferramenta em questão, como por exemplo a utilização de padrões, visto que melhorias devem ser feitas antes que ela seja disponibilizada para uso de terceiros.

Assim sendo, este trabalho tem o objetivo de realizar a Reengenharia de um software acadêmico, de modo coerente com a proposta dos autores citados, onde há uma considerável melhora de manutenibilidade e uma evolução de funcionalidades do *software*, sem que o seu foco de atuação seja alterado.

2 Conceitos

2.1 Conceitos de Reengenharia de Software

Ao ser aplicada, a RS pode evidenciar descuidos que passaram despercebidos em um projeto, tais como a falta de legibilidade de um código-fonte, ou a falta de organização da comunicação entre módulos desse projeto, bem como uma documentação mal feita e/ou pouco eficaz, de modo a penalizar o rápido entendimento e a percepção do funcionamento do sistema como um todo.

Em casos como estes, onde é identificada a existência de uma estrutura desorganizada, acaba-se adotando, na maioria das vezes, algum tipo de padronização. Ela pode ser adotada tanto para estruturas de mais alto nível de abstração, como por exemplo, para realizar o remanejamento de classes em pacotes acrescentando um novo fluxo de comunicação entre elas, quanto para estruturas de mais baixo nível de abstração, onde pode haver, por exemplo, a adoção de palavras-chave em nomenclaturas de variáveis e/ou métodos. Assim sendo, vê-se a necessidade de utilização de outro conceito da ES, a Arquitetura de Software (AS).

A AS, é um conceito não muito antigo da ES. Ela é mencionada pela primeira vez ao final do ano de 1969, em um relatório escrito por vários autores, chamado *Software Engineering Tecnhiques*, e atualmente é caracterizada como "uma visão abstrata entre relacionamentos de elementos que compõem um sistema, bem como da topologia desempenhada por eles" (SILVA, 2000; GERMOGLIO, 2009).

Segundo BUSCHMANN (1996), os padrões utilizados na AS constituem uma organização estrutural fundamental dos esquemas de software, especificam as responsabilidades e incluem regras e diretrizes das relações entre eles. E embora seja indispensável a utilização de padrões na AS, um único padrão por si só, não define uma AS por completo, pois a resolução de um problema em si, pode não resolver o conjunto de problemas globais que devem ser solucionados com essa arquitetura.

2.1.1 Padrões Arquiteturais

Padrões nada mais são do que propostas de soluções convergentes para problemas comuns. A AS faz uso de padrões para compor modelos para diferentes papéis, incorporados em múltiplas visões.

Os padrões utilizados durante o processo de criação da arquitetura de um *software* recebem o nome especial de Padrões Arquiteturais e são definidos como *templates* que concretizam a AS. Eles especificam a estrutura principal de uma aplicação e todo o seu desenvolvimento (ou redesenvolvimento) é regido por essa estrutura. Cada padrão dá sua parte de contribuição para que propriedades globais específicas da aplicação sejam alcançadas, visto que estas são necessárias para o sucesso almejado no desenvolvimento de software.

Ainda na visão de BUSCHMANN (1996), esses padrões podem ser divididos em um conjunto de quatro categorias. São elas:

1. **Estruturais (*From Mud Structure*)**: Apoiam uma tarefa geral de um sistema em subtarefas. Os padrões que se enquadram nessa categoria são:
   (a) Padrão de Camadas
   (b) Padrão de *Pipe* e Filtros
   (c) Padrão de Quadro-Negro
2. **Sistemas Distribuídos**: Compreende macro-soluções para sistemas baseados em distribuição. O único padrão que se enquadra a esta categoria é o seguinte:
   (a) Padrão Broker
3. **Sistemas Interativos**: Compreende dois padrões que suportam a estruturação de sistemas de software que trabalham na iteração homem-máquina:
   (a) Padrão Model-View-Controller (MVC)
   (b) Padrão Presentation-Abstraction-Control (PAC)
4. **Sistemas Adaptativos**: Suportam a extensão de aplicações e suas adaptações às tecnologias em evolução, bem como às mudanças dos requisitos funcionais. Os dois

padrões compreendidos nessa categoria, são:

(a) Microkernel

(b) Reflection

Dos padrões arquiteturais citados anteriormente, pode-se citar que o MVC foi implementado no módulo *web* (desenvolvida na linguagem de programação PHP[1]) do software WSDS, o que sustenta a manutenibilidade dessa parte do sistema desde os primórdios de sua construção. Essa estrutura *web* pode ser observada na figura 2.2 e será detalhada mais adiante no tópico 2.2.2.

2.1.2 Padrões de Projeto

Os padrões de projeto foram criados para descrever um conjunto de soluções que já foram testadas e que são recorrentes em desenvolvimento de software que utilizam o paradigma de Orientação a Objetos (OO) em sua construção. Neste capítulo são descritos os padrões de projeto do tipo *Gang Of Four* (GoF), sendo alguns deles utilizados na implementação da ferramenta WSDS.

Padrões de projeto são descrições de objetos e classes comunicantes que precisam ser personalizadas para resolver um problema geral de projeto num contexto particular.

Atualmente, as técnicas dessa categoria de padrões são divididas em duas vertentes. A primeira, conhecida como GoF, foi pesquisada e descrita pelos autores do livro *Design Patterns: Elements of Reusable Object-Oriented Software* (GAMMA, 1994). Ela elicita soluções abstratas a problemas comuns da orientação objetos e garante que exista ao menos uma solução testada empiricamente, caso sejam seguidas todas as instruções contidas em determinado padrão do GoF. Ao todo, são vinte e três padrões ou soluções.

Já a segunda, chamada *General Responsibility Assignment Software Patterns* (GRASP), foi introduzida pelo autor do livro *Applying UML and patterns* (LARMAN, 2002). Ao contrário da primeira, esta não possui respostas prontas e elucida princípios que atingem um alto grau de abstração e que, na maioria das vezes, envolvem a UML (*Unified Modeling Language*). Ao todo, são nove padrões.

[1]PHP: O *Hypertext Preprocessor* é uma linguagem de programação interpretada e livre, que possibilita a criação de páginas dinâmicas e possui grande número de utilizadores.

Em geral, a maioria dos padrões da técnica GoF, pode ser vista como uma especialização de uma combinação de poucos outros padrões da técnica GRASP. Este fato torna o GoF uma técnica com nível de especificidade mais elevada e, como consequência, o número de padrões é maior. Em virtude de sua maior objetividade, a GoF possui maior utilização no meio computacional.

A seguir, são descritos sucintamente os padrões de projeto do tipo GoF.

1. **Padrões de Criação**: Têm como principal ponto comum a abstração do modo de instanciação, composição e representação de classes. Sua principal vantagem é a flexibilização do **que**, **quando** e **como** é criado e **quem** a cria.

   (a) ***Abstract Factory***: Este padrão é também conhecido como "Kit". Para utilizar-se dele, um sistema deve ser configurado como um produto de uma família de múltiplos produtos, ou então, deve ser um sistema portável entre vários estilos de aparência. Além disso, este padrão especifica um método Factory que fabrica objetos de um tipo particular.

   Como exemplo de sua utilização, pode-se citar uma aplicação qualquer que faz o uso de diferentes *skins*, ou seja, onde a preferência de leiaute utilizada por um usuário pode ser modificada no momento em que ele desejar.

   (b) ***Builder***: Este padrão é geralmente utilizado em casos que se tenham uma entrada e duas ou mais saídas, ou seja, quando o processo de criação de um objeto pode gerar diversas representações. Além disso, este padrão separa a construção de um objeto complexo de sua representação.

   Como exemplo de sua utilização, pode-se citar uma aplicação qualquer de texto ou áudio, que deva manter em aberto as possibilidades de formatos exportados. Ou seja, na Orientação a Objetos (OO), seria uma aplicação onde uma classe genérica fosse central, e outras classes filhas partiriam desta, e o número dessas classes filhas fosse ilimitado.

   (c) ***Factory Method***: Este padrão transfere a responsabilidade de instanciação de classes às subclasses. Ou seja, permite adiar a instanciação de classes em tempo de execução, o que não é previsto no paradigma de OO.

Como exemplo de sua utilização, pode-se citar uma aplicação qualquer onde são criados documentos, em que o formato não é conhecido. Ou seja, em aplicações onde documentos genéricos são criados, abertos ou fechados. Este padrão tem ampla utilização em *Frameworks*, e utiliza classes abstratas como um meio de passagem.

(d) ***Prototype***: Este padrão especifica uma forma de se obter uma instância-protótipo e novas instâncias copiadas desse protótipo.

Como exemplo de sua utilização, pode-se citar uma aplicação qualquer de desenho que possibilite a criação e composição de novos objetos, a partir de objetos fundamentais.

(e) ***Sigleton***: Garante que só exista uma instância única de determinada classe, e permite o acesso global da mesma.

É amplamente utilizado entre conexões de aplicações e seus bancos de dados, pois garante que uma conexão não seja duplicada, e consequentemente, um caminho único de comunicação de dados seja estabelecido.

2. **Padrões Estruturais**: São utilizados para combinar diferentes tipos classes, de modo que seja gerada uma nova classe com propriedades das classes ancestrais.

(a) ***Adapter***: Este padrão é também conhecido como "Wrapper". Ele permite que interfaces incompatíveis trabalhem em conjunto.

Como exemplo de sua utilização, pode-se citar uma aplicação qualquer onde a vazão de água seja medida pelo fluxo de água que atravessa um paralelepípedo reto, passe a ser medida pelo fluxo que atravessa um cilindro curvilíneo, com dimensões menores do que as do sólido anterior.

(b) ***Bridge***: Este padrão é também conhecido como "Handle/Body". Ele permite que haja desacoplamento entre uma abstração e sua implementação.

É amplamente utilizado em aplicações que devem ser utilizadas em mais de uma plataforma, como por exemplo, um aplicativo genérico que deva rodar em

IOS[2], Android[3] e Windows Phone[4].

(c) ***Composite***: Este método define a composição de classes em formato de árvore, onde a hierarquia seja da forma parte-todo.

Como exemplo de sua utilização, podem-se citar aplicações gráficas, onde um objeto possa ser incorporado dentro de outro objeto, semelhente ao primeiro, como em um jogo de espelhos.

(d) ***Decorator***: Este padrão tem como principal objetivo agregar funções a uma classe um objeto, através da utilização de subclasses.

Como exemplo de sua utilização, pode-se citar uma aplicação qualquer de visualização de documentos, com dimensões fixas, que não tem os componentes de barras de rolagem, mas que deverá passar a tê-los.

(e) ***Façade***: Este padrão define uma interface com nível de abstração elevado, que unifica interfaces de subsistemas. Com isso, adquire o sistema adquire maior usabilidade.

Como exemplo de sua utilização, pode-se citar uma aplicação com diversos métodos de ordenação numérica, onde uma classe central concentra a chamada de cada um deles.

(f) ***Flyweight***: Este padrão permite criação de objetos em massa, ou seja, de granularidade fina.

Como exemplo de sua utilização, pode-se citar um editor textual genérico, baseado em OO, onde cada caractere fosse instanciado como um objeto. Ao invés de cada objeto conter todas as características como cor, tipo de fonte e localização, este objeto só tem a responsabilidade de guardar a letra ou número que representa. Já as outras informações citadas, são guardadas por outro objeto, de forma que se diminua a quantidade de memória destinada à aplicação.

(g) ***Proxy***: Este padrão é também conhecido como "Surrogate". Ele fornece um marcador de localização de outro objeto, para controlar o seu acesso.

[2]IOS: Sistema operacional móvel da empresa Apple inc.
[3]Android: Sistema operacional móvel da empresa Google inc.
[4]Windows Phone: Sistema operacional móvel da empresa Microsoft corporation.

Como exemplo de sua utilização, pode-se citar a classe que guarda informações da cor, fonte e localização de um caractere do exemplo anterior, relativo ao padrão *Flyweight*.

3. **Padrões Comportamentais**: Estes padrões se preocupam com a comunicação entre classes e objetos, bem como a organização de seus algoritmos.

   (a) ***Chain Of Responsability***: Este padrão permite o não acoplamento de solicitações entre um emissor e um receptor.

   Como exemplo de sua utilização, pode-se citar uma aplicação qualquer, que possui um botão de ajuda durante todo o seu tempo de execução. Sempre na mesma posição, este botão pode apresentar diferentes mensagens, dependendo de que parte da aplicação o usuário se encontra no momento em que o aciona.

   (b) ***Command***: Este padrão permite a criação de ações que podem ser desfeitas, com a utilização de enfileiramento de chamadas ou *log*.

   Como exemplo de sua utilização, pode-se citar uma aplicação qualquer, que possui os métodos de copiar, recortar e colar textos. Ao se chamar qualquer um desses métodos, é empilhada uma invocação de método, que pode ser desfeita pelo usuário quando ele desejar.

   (c) ***Interpreter***: Este padrão permite criar gramáticas para uma linguagem simples, representar sentenças nessa linguagem e, consequentemente, interpretá-las.

   Como exemplo de sua utilização, pode-se citar uma aplicação que ao tentar localizar padrões em textos, através de vários algoritmos customizados de comparação de caracteres, faz o uso de algoritmos de busca, que interpretam uma expressão regular, de modo mais abrangente.

   (d) ***Iterator***: Este padrão é também conhecido como "Cursor". Ele fornece uma interface que percorre diferentes tipos de objetos, fornecendo poucas funções para se iniciar e finalizar um percurso em uma lista, de forma sequencial. Diferentes objetos iteradores apresentam as mesmas funções padrões, que podem desempenhar tarefas diferentes, porém com a mesma finalidade, e interagir com

diferentes tipos de listas (objetos).

Como exemplo de sua utilização, pode-se citar uma biblioteca orientada a objetos qualquer que percorra listas com iteração polimórfica, ou seja, onde há diferentes objetos agregados[5].

(e) ***Mediator***: Este padrão permite que objetos deixem de se referenciar uns aos outros, de forma a diminuir o acoplamento entre eles. Para tanto, é definido um objeto do tipo Mediator, que promove uma forma independente de iteração.

Como exemplo de sua utilização, pode-se citar uma estrutura de objetos qualquer, onde os objetos só devem conhecer outros objetos que são necessários para seu funcionamento, ou seja, onde não haja um conhecimento geral de objetos, onde um objeto consegue visualizar qualquer outro objeto do projeto.

(f) ***Memento***: Este padrão é também conhecido como "Token". Ele fornece a possibilidade de se extrair o estado interno de um objeto, sem violar o seu encapsulamento.

Como exemplo de sua utilização, pode-se citar qualquer editor gráfico genérico que permita a conexão de objetos, onde em determinado momento, o usuário pode querer recorrer a um estado anterior ao atual. Ou seja, ele pode querer desfazer uma ou mais movimentações.

(g) ***Observer***: Este padrão é também conhecido como "Dependents". Ele fornece meios de se obter uma relação de objetos de um para muitos, que faz com que ao se modificar o estado de um objeto, todos os seus dependentes sejam notificados e atualizados de forma automática.

Como exemplo de sua utilização, pode-se citar alguns objetos genéricos, que possuem elementos gráficos e que dependem de outro objeto que armazena dados matriciais matemáticos. Ou seja, quando os dados matemáticos são alterados, os gráficos devem ser alterados automaticamente.

(h) ***State***: Este padrão é também conhecido como "Objects for States". Ele fornece meios para que um objeto altere o seu comportamento, quando o seu

[5]Objetos Agregados: Em orientação a objetos, representam objetos que são compostos por várias partes de outros objetos.

estado interno é modificado.

Como exemplo de sua utilização, podem-se destacar, implementações em que realmente há alterações de estado (como é sugerido pelo nome do padrão) de um objeto, como o passar do modo ativo para inativo, ou de conexão estabelecida para perdida e vice-versa. Em suma, sua utilização é recomendada em casos de conexões de dados, como entre a comunicação de um navegador web e um servidor, através dos protocolos TCP e/ou UDP, ou então entre a comunicação de um cliente de banco de dados e um servidor, e assim por diante.

(i) ***Strategy***: Este padrão é também conhecido como "Policy". Ele permite com que um objeto decida qual algoritmo utilizar dentre uma gama de opções de objetos. Ou seja, dentre objetos semelhantes, que desempenham a mesma função mas utilizando caminhos diferentes, o objeto invocador escolhe o que melhor lhe serve, em tempo de execução.

Como exemplo de sua utilização, pode-se citar um objeto de uma planilha de texto que invoca diferentes outros objetos para ordenar células de maneiras diferentes, dependendo do número de células, para obter sempre o melhor desempenho de ordenação.

(j) ***Template Method***: Este padrão permite a criação de carcaças de algoritmos, onde alguns passos são adiados para serem implementados em subclasses.

É utilizado, por exemplo, em *frameworks* que lidam com a abertura e escrita de documentos, possibilitando a abertura de diversos tipos de extensões de arquivos e diversas orientações de escrita.

(k) ***Visitor***: Este padrão permite que novas operações sejam definidas sem que as classes que interoperam sejam alteradas.

Como exemplo de sua utilização, pode-se citar qualquer compilador genérico que faça uso de uma Árvore Sintética Abstrata (AST), onde há verificações inicialização de variáveis e geração de códigos executáveis.

2.2 Contextualização da Aplicação (Simulador)

O WSDS, *software* desenvolvido pelo pesquisador Ciro B. Barbosa, da Universidade Federal de Juiz de Fora (UFJF), foi criado no ano de 2011 com o intuito de possibilitar a simulação computacional de Sistemas Dinâmicos, de forma interativa e concorrente, em um ambiente *web* onde usuários pudessem criar seus próprios modelos textuais, e visualizar a variação dos parâmetros do modelo aplicado em função do tempo, dentro do período de simulação(BARBOSA et al, 2011).

Para tal, foi utilizada a técnica de Dinâmica de Sistemas, ao invés de Sistemas de Equações Diferenciais Ordinárias (SEDOs). A principal vantagem dessa técnica, em relação aos SEDOs, é não ser necessário um conhecimento avançado de matemática. Ao invés disso, ela permite identificar conceitos e estabelecer suas relações de forma quantitativa, usando apenas equações algébricas.

2.2.1 Dinâmica de Sistemas

Desenvolvida inicialmente como um método de estudo de gerenciamento de complexos sistemas industriais em 1956 por Jay W. Forrester, e implementada por ele como uma disciplina de modelagem no *Massachusetts Institute of Technology* (MIT), em 1961, a Dinâmica de Sistemas (DS) expandiu-se a outras áreas ao longo das décadas, tais como, a urbanização, economia, medicina, ciência e educação (FORRESTER, 1996). Além de ser um método de modelagem que pode ser detalhado em uma simples folha de papel através de símbolos padrões, no mundo contemporâneo, ela se une à Simulação Computacional (SC), e permite a visualização em tempo real do resultado de inúmeras intervenções realizadas em determinado modelo[6] ao longo de sua execução, mostrando novas formas de se atacar um problema abordado (JÚNIOR et al, 2006).

Para MORECROFT (2000), a simplicidade gráfica aliada a força analítica da DS desenvolvida por Forrester, quando aplicada corretamente, eleva o entendimento de um modelo de forma considerável. Entretanto, para se obter bons resultados em uma SC, deve-se ter o auxílio de ferramentas de propósito computacional.

[6]Modelo: Em Dinâmica de Sistemas, um modelo nada mais é do que uma descrição matemática formal de um sistema como um todo. Explicita inter-relações entre vários artefatos.

A figura a seguir, mostra os elementos básicos da DS:

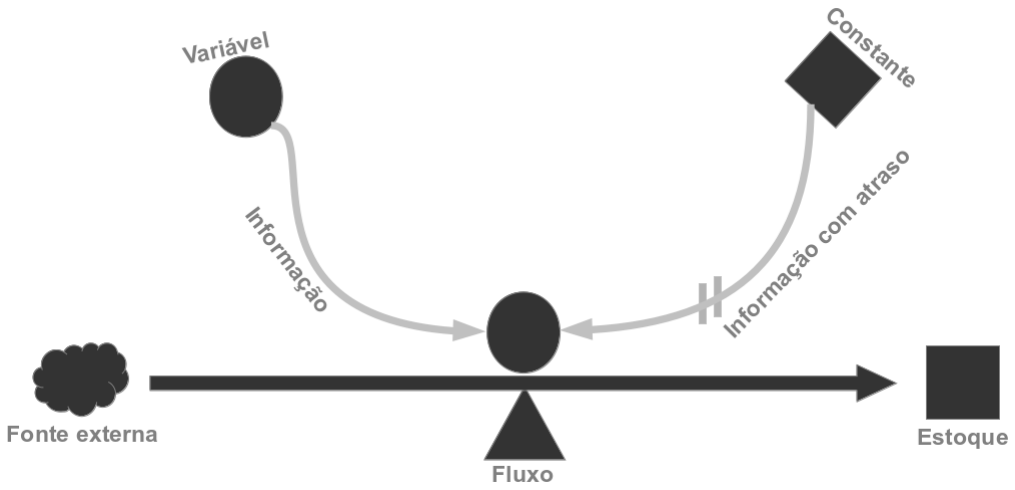

Figura 2.1: Elementos da Dinâmica de Sistemas

Existem algumas ferramentas conhecidas de DS, porém poucas são gratuitas. Em geral, quase todas possuem uma versão *trial* (gratuita por certo tempo e com funcionalidades restritas) e outra *full* (paga e com funcionalidades irrestritas). As mais conhecidas são: Forio (FORIO, 2012), Powersim (POWERSIM, 2012), Stella (STELLA, 2012), Vensim (VENSIM, 2012) e Wlinkit (WLINKIT, 2012).

Há também trabalhos relacionados na área, como por exemplo, o do pesquisador, Márcio de Oliveira Barros, que descreve a especificação e implementação de um compilador (no software *Hector*) e um ambiente de simulação (*Illium*) como ferramentas distintas. Uma de suas importantes contribuições, que é utilizada no WSDS por exemplo, é a linguagem de simulação textual (LST), que com sua criação, permitiu a substituição da sintaxe gráfica (inata à DS) por uma sintaxe textual, gerando assim, modelos mais concisos (BARROS, 2001).

De modo a por em prática o fácil entendimento desta linguagem formal de Dinâmica de Sistemas, definida pelo pesquisador acima citado, foi criado o WSDS, tendo como principal vantagem às ferramentas desenvolvidas por ele, a compilação e simulação de sistemas dinâmicos de forma sincronizada e em um único ambiente. Sua aplicabilidade e robustez serão abordadas no tópico a seguir.

2.2.2 Arquitetura do Simulador Web de Dinâmica de Sistemas

O simulador WSDS utiliza a arquitetura cliente-servidor. Pode-se dizer que ela configura uma estrutura onde processos clientes interagem com processos servidores a fim de acessar recursos compartilhados que os servidores gerenciam (COULOURIS, 2007). Ou seja, este modelo permite a existência e comunicação entre um ou mais clientes com um ou mais servidores, onde toda a comunicação entre clientes e servidores deve trafegar ao menos através de um servidor. Atualmente, esta aplicação é executada em um único servidor, de forma centralizada. Mas este fato não impede que inúmeros clientes a acessem de forma simultânea.

A figura 2.2, mostra o desenho arquitetural do software WSDS.

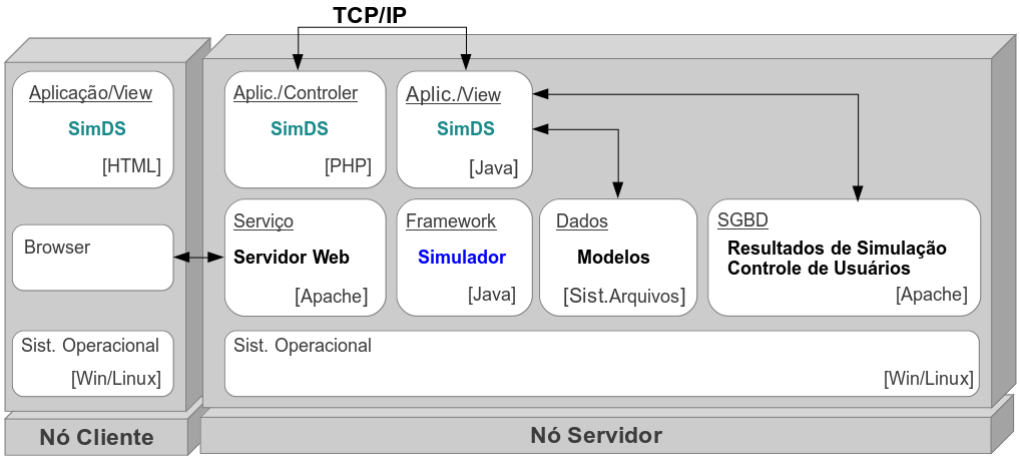

Figura 2.2: Arquitetura do software WSDS

Ao lado esquerdo, externo ao servidor, podem-se visualizar 2 (duas) camadas da aplicação propriamente dita que compõem o "Nó Cliente", ou seja, o *Front-End*. São elas:

- O ***Browser***: que é o meio em que a aplicação interage com o usuário.
- E a ***View***: que é uma instância da parte visual do simulador em si tida pelo usuário, e que utiliza o *Browser* como um meio de comunicação com o serviço web, Apache[7]. Sua principal função é mostrar informações amigáveis ao usuário.

[7]Apache: Desde Janeiro de 1996, em que sua representatividade era de 20%, até os dias atuais, em que passou a ter 55% de representatividade, o Apache se tornou o servidor web mais utilizado em todo o mundo. Fonte: http://news.netcraft.com/

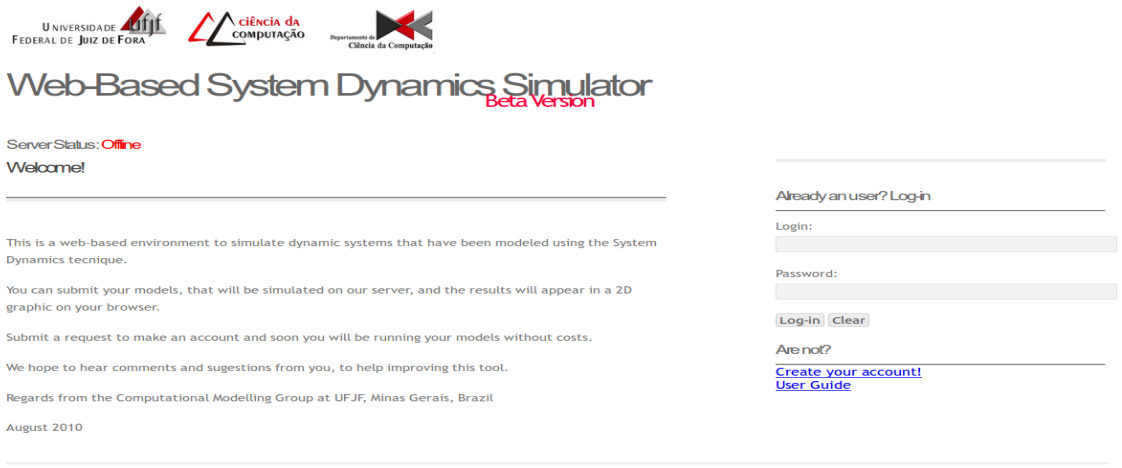

Figura 2.3: *View* principal do simulador WSDS: *homepage*.

Já no bloco à direita, existem outras camadas que compõem o "Nó Servidor"do simulador. São elas:

- O ***Controler***: que é o módulo escrito na linguagem de programação PHP, que controla as requisições do usuário. Sua principal função é monitorar as informações, transformar eventos em requisições de serviços e encaminhá-las ao destino correto, que pode ser a *View* ou o *Model*.

- O ***Model***: é o responsável por encapsular os dados do núcleo da aplicação. Contém procedimentos específicos da aplicação.

- **Serviço**: que é responsável pela execução do serviço web em si. Encarregado de processar solicitações HTTP, HTTPS, FTP, entre outras, e encaminhar as devidas respostas com conteúdo ao usuário. Possui boa performance e é capaz de executar códigos-fonte em diversas linguagens de programação, tais como: PHP, Perl, Shell Script e ASP. Além disso, também possui uma vasta gama de recursos, como por exemplo, a criptografia de dados, autorização de acessos, geração de logs de erros e suporte a proxy.

- ***Framework***: é considerado o carro-chefe do simulador, este módulo contém toda a lógica e compilação de dados referentes à DS.

- **Dados**: que é o módulo responsável por salvar e exportar os modelos gerados pelo *Model*.

- **SGBD**: responsável por interagir com o *Model* e posteriormente, guardar todas as informações dos usuários e suas simulações.

Apesar de possuir a opção de ser executado em modo stand-alone, onde o administrador do sistema pode inserir os dados de sua simulação diretamente no servidor, o simulador WSDS têm seu foco na execução simultânea, de forma on-line, conforme dito anteriormente.

Após efetuar login na ferramenta, o usuário tem acesso a uma área pessoal, onde são guardados todos os modelos e meta-modelos que desenvolveu (figura 2.4). Os códigos-fonte do modelo e meta-modelo apresentados nesta figura, podem ser visualizados no apêndice **A.1**, e a sua modelagem em DS, no apêndice **A.2**.

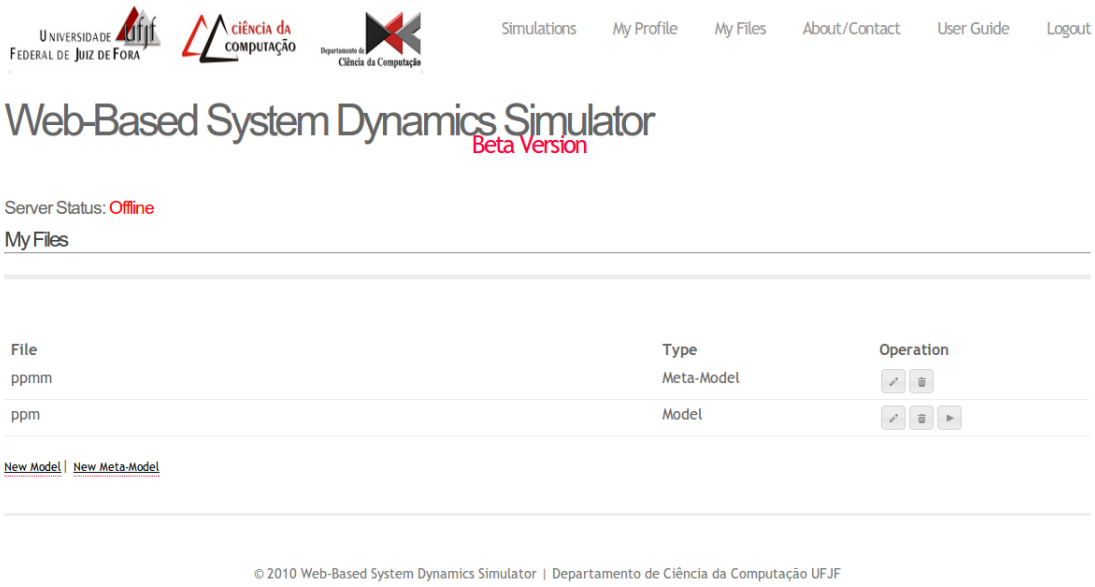

Figura 2.4: Espaço de trabalho pessoal do WSDS

Ao validá-los, é disponibilizada ao usuário uma nova função, na verdade a principal, que é a de simular a iteração entre esses modelos e meta-modelos. Para tanto, deve-se inserir os parâmetros de entrada da simulação que são: o passo de simulação (**delta T**), um valor para retardar o tempo de simulação e possibilitar a iteração do usuário (**delay**) e **número de iterações** (figura 2.5).

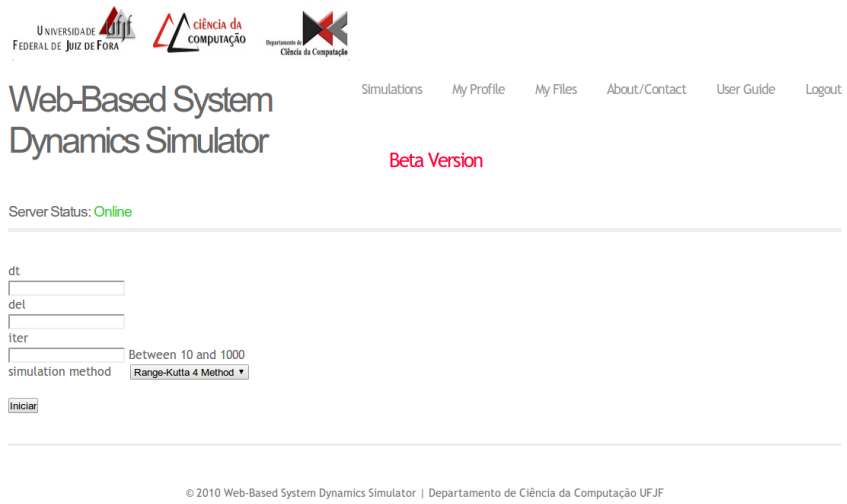

Figura 2.5: Parâmetros de entrada de um exemplo de simulação do WSDS

Os resultados de uma simulação, então podem ser visualizados em uma janela como a seguinte.

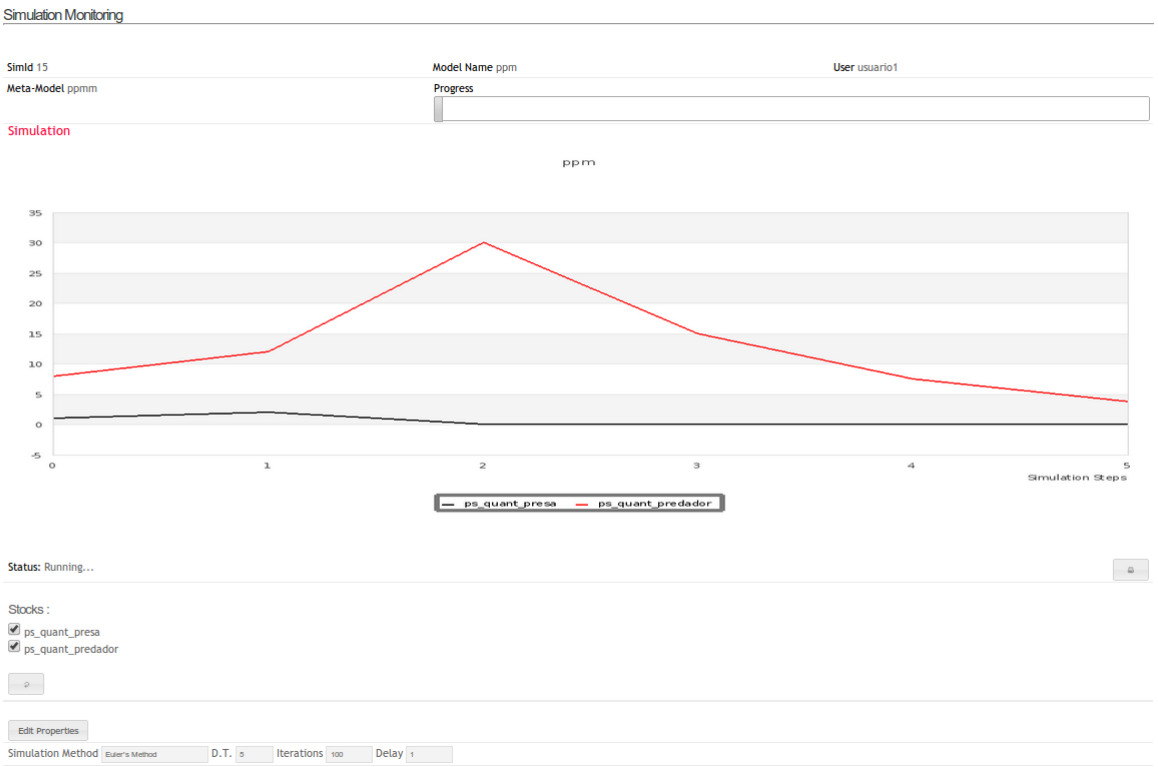

Figura 2.6: Execução de um exemplo de simulação no WSDS

Como pode ser observado, o WSDS possui o conjunto de funcionalidades necessárias para ser operacional, porém, a lista de escolha do método de simulação visualizada na figura 2.5, não havia sido implementada antes deste trabalho. E para lançar mão de uma nova importante funcionalidade como esta, vê-se a necessidade de utilização dos

padrões de projeto, visto que não existem limites de métodos de simulação que podem ser empregadas no simulador.

Portanto, no próximo capítulo são elucidados casos como este, onde estes padrões são empregados no *software* WSDS.

3 Desenvolvimento

Através da visualização da arquitetura do sistema WSDS, elucidada no tópico anterior, nota-se que há uma variedade maior de camadas que constituem o "Nó Servidor" do que o "Nó Cliente". E dentre elas, destacam-se o **Model** e o **Framework**, pois são parte vital do *software* já que são encarregadas de realizar todos os cálculos e projeções da DS. Devido a essa maior importância e complexidade, os esforços do trabalho de RS foram centralizados nesses módulos.

Porém, outros módulos como a *View* e o SGBD, apresentavam erros antes deste trabalho, e atrapalhavam a experiência com o usuário. Dentre algumas alterações que podem ser citadas, foram bloqueados os botões de simulação do usuário quando o servidor está *off-line*, foi bloqueada a opção de editar as propriedades de uma simulação que está em estado diferente de ***Running***, e campos de tabelas de banco de dados tiveram que ser corrigidos.

3.1 Padrões aplicados

A construção de vários módulos que interagem entre si, fez com que o simulador WSDS contivesse mais de 10.000 (dez mil) linhas de código-fonte e adquirisse um porte médio de *software*. Alguns de seus módulos foram desenvolvidos em momentos diferentes e desenvolvedores não tiveram uma relação direta de trabalho. Além disso, também não houve motivação para que fossem adotadas na época certas padronizações. Estes fatos entram em conflito com alguns dos principais atributos de produtos de software que garantem a qualidade do mesmo, dentre elas, a **manutenibilidade**, a **eficiência** e a **confiança** (SOMMERVILLE, 2001).

Antes da elaboração deste trabalho foram observadas oportunidades de melhoria do WSDS, dentre as quais, algumas permitiram a remodelagem através de padrões de projeto, de acordo com o conjunto de padrões do grupo GOF introduzido na seção 2.1.2, e com isso, contribuíram com a melhoria de qualidade do *software*. São eles:

3.1.1 Padrão *Bridge*

A **versatilidade** é uma característica que faz falta a *softwares* que devem ser portáveis a vários sistemas operacionais. Mas também é importante naqueles que esperam agregar, em um futuro próximo, diferentes funções que desempenham papéis semelhantes e que têm a mesma atividade fim, ou seja, quando o mesmo resultado é esperado por diferentes implementações. Nesses casos, o padrão Bridge é indicado, pois desacopla objetos e facilita a **versatilidade**, que é uma característica, quanto a manutenibilidade, que é um atributo de software.

No caso do simulador, houve a possibilidade de aplicar esse conceito em um caso especial, onde imaginou-se um dia dar a opção ao usuário de alternar entre vários métodos de simulação antes de executar suas simulações, possibilitando a escolha entre custo/benefício, tempo de execução/precisão.

Anteriormente à aplicação deste padrão, toda a lógica do único método de simulação possível estava acoplado à classe MetodoSimulacao, da seguinte forma:

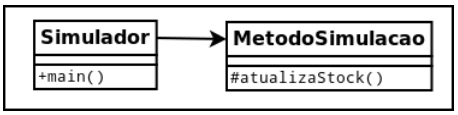

Figura 3.1: Diagrama de Classes: Classe MetodoSimulacao sem o Padrão *Bridge*.

Com isso, ao serem implementados novos métodos de simulação, seria inevitável a inserção de vários operadores condicionais (if..then..else), que acarretariam, neste caso, em um código-fonte pouco manutenível. Ou seja, para que um desenvolvedor saiba que nesta classe existem vários métodos de simulação implementados, ele deve entrar em cada condição possível ao invés de simplesmente visualizar um simples diagrama de classes, que ajudaria muito na manutenibilidade.

Porém, ao utilizar-se o padrão citado, houve ganho tanto na manutenibilidade quanto na versatilidade do código, visto que foi possível um maior desacoplamento entre objetos, e com isso, ao ser necessária a utilização de um novo método de simulação, basta criar uma classe que a implemente, e incluir uma única linha de chamada na classe **Simulador** passando o seu próprio nome como parâmetro, que esta será executada transparentemente pelo framework do simulador, e posteriormente será exibida na *View* de

simulação.

Como resultado, obtém-se a seguinte estrutura de classes:

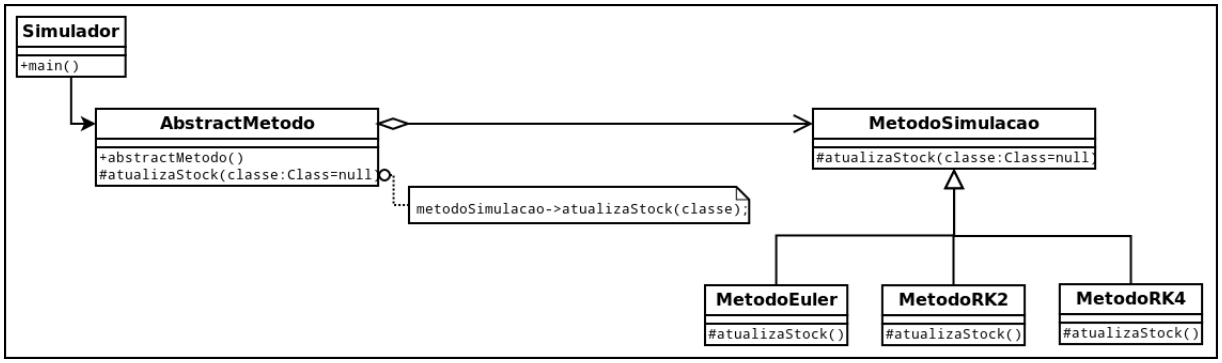

Figura 3.2: Diagrama de Classes: Classe MetodoSimulacao com o Padrão *Bridge*.

3.1.2 Padrão *Visitor*

A **otimização** é outra característica marcante que deve ser observada em *softwares* que não têm um único núcleo, ou seja, que fazem uso de vários módulos que se comunicam durante toda a sua execução. E, para garantir essa característica, além da manutenibilidade, deve-se lançar mão de outro atributo de software, a **eficiência**.

O padrão Visitor consegue proporcionar tanto a manutenibilidade quanto a eficiência, relativa a processamento e armazenamento de dados, em casos onde são montadas grandes estruturas de dados que devem ser consultadas inúmeras vezes durante a execução de uma única instância. Ou seja, em casos como o do WSDS, em que um compilador próprio deve ser utilizado e onde checagens de tipos e valores devem ser realizadas a todo instante.

Especificamente no simulador abordado, antes deste trabalho, não havia uma estrutura de dados definida separadamente, e toda verificação de atributos e valores era feita dentro de um laço, ou seja, em uma única estrutura de repetição. E com isso, as análises sintática e semântica apresentavam estado indistinguível uma da outra, o que diminuía significativamente sua manutenibilidade. Além disso, com esta estrutura em bloco, para se consultar valores de variáveis, inúmeras buscas por palavra-chave eram feitas, percorrendo toda essa estrutura, o que tornava este código, nada otimizado, em pouco eficiente.

Para garantir a manutenibilidade das principais classes do framework do simulador, o visitor foi implementado, gerando a seguinte estrutura:

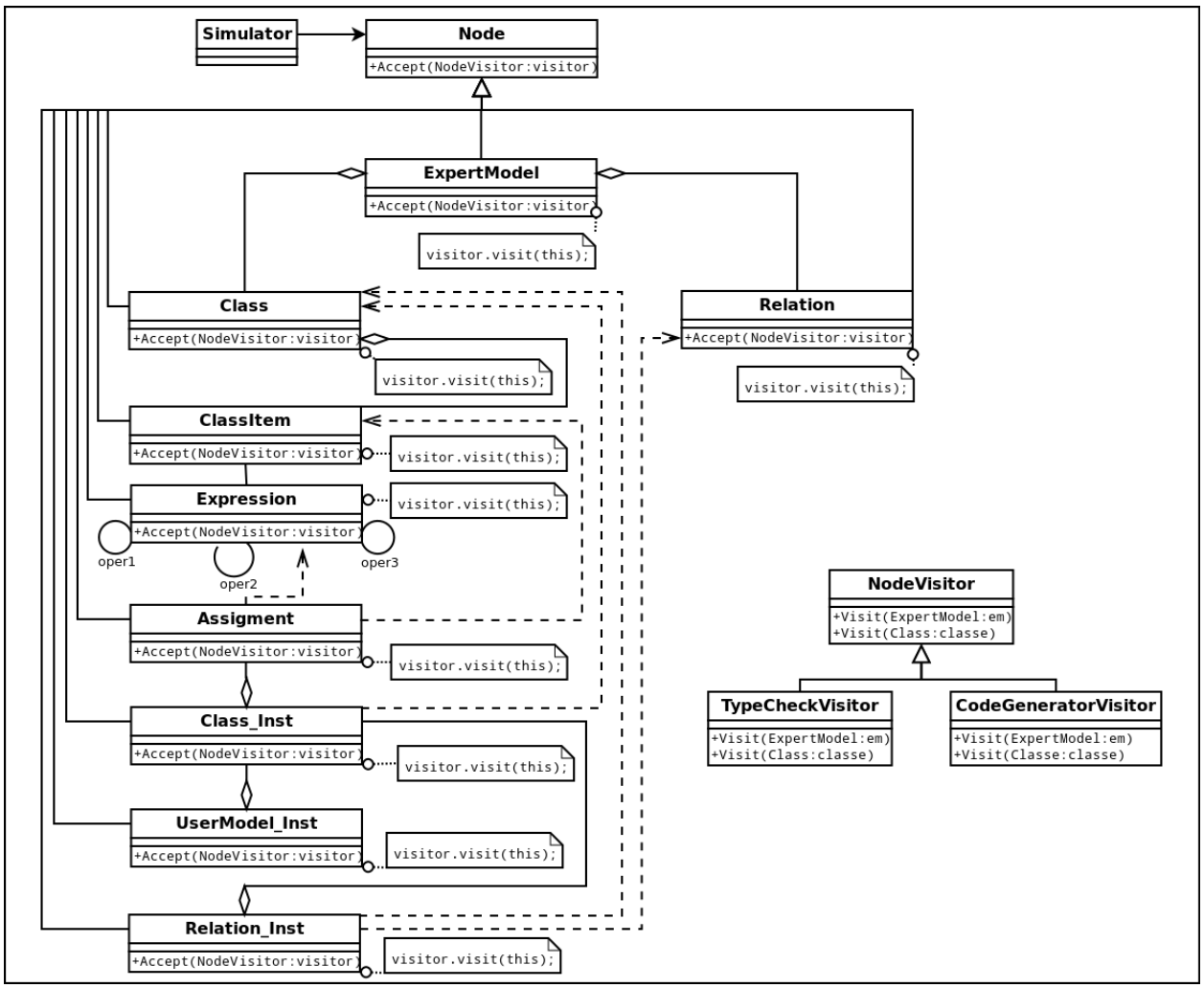

Figura 3.3: Diagrama de Classes: Classe Node com o Padrão *Visitor*.

Com isso, gera-se uma nova estrutura, em forma de árvore, a Árvore de Sintaxe Abstrata (AST), que servirá como base para as análises sejam feitas separadamente, melhorando a manutenibilidade do framework (LOURDEN, 2004).

Outra vantagem, é que agora, após a aplicação do visitor, é possível se implementar outra estrutura exclusivamente para armazenar variáveis e seus valores de modelos e meta-modelos, de modo a melhorar significativamente a sua eficiência. Esta nova estrutura é a Tabela de Símbolos que poderá ser do tipo Lista Encadeada ou do tipo Tabela *Hash* (AHO, 1995).

3.1.3 Padrão *Singleton*

A otimização também faz parte deste padrão. Da mesma forma que o padrão anterior, o *singleton* proporciona ganhos no armazenamento de dados, visto que a sua principal função é garantir a unicidade de determinado objeto. Ou seja, ele garante que este objeto seja instanciado uma única vez durante a execução do *software* em questão, poupando a memória física.

Esta unicidade de objetos, quando bem empregada, além de gerar um ganho de **eficiência**, gera também outro ganho que é a **confiança**, visto que o singleton é empregado na maioria das vezes em classes que possuem comunicação de dados, e para isto, utilizam portas de comunicação. Em suma, não são abertas portas nem criados objetos extras desnecessariamente. Especificamente no simulador WSDS, o *singleton* foi utilizado em uma classe que estabelece a comunicação de dados. E a partir de então, sempre que necessário, o objeto de conexão com o banco é retomado para fazer um tipo de consulta. Sua estrutura é mostrada a seguir:

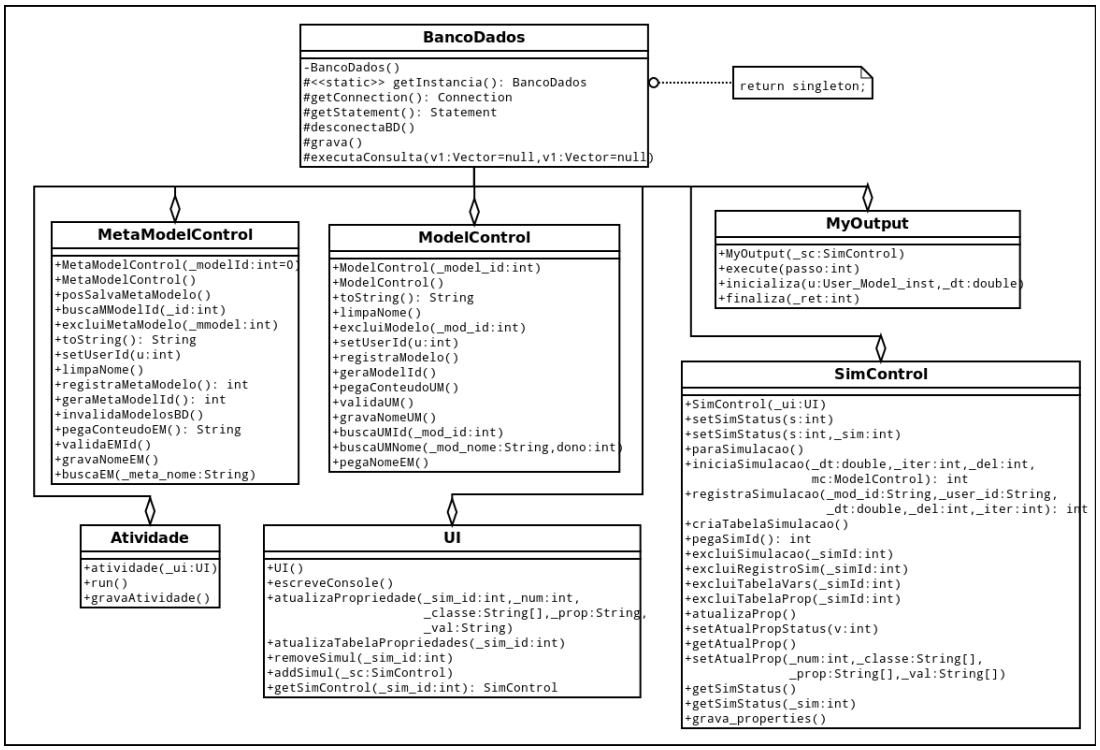

Figura 3.4: Diagrama de Classes: Classe BancoDados com o Padrão *Singleton*.

3.2 Novas funcionalidades

O servidor web multithread do simulador, que antes apresentava instabilidade e *delay* quando vários usuários executavam suas simulações simultaneamente, agora apresenta grande estabilidade e consegue executá-las quase que instantaneamente. Isso se deve a remodelagem ocorrida na classe "Comunica" do *Model* do simulador. Embora ela tenha sido desenvolvida como um *server socket*, estava utilizando apenas um canal de comunicação, fazendo com que os clientes entrassem em fila de espera para serem "ouvidos" e terem suas simulações iniciadas(SOURCEFORGE.NET, 2013).

A figura seguinte, ilustra essa situação.

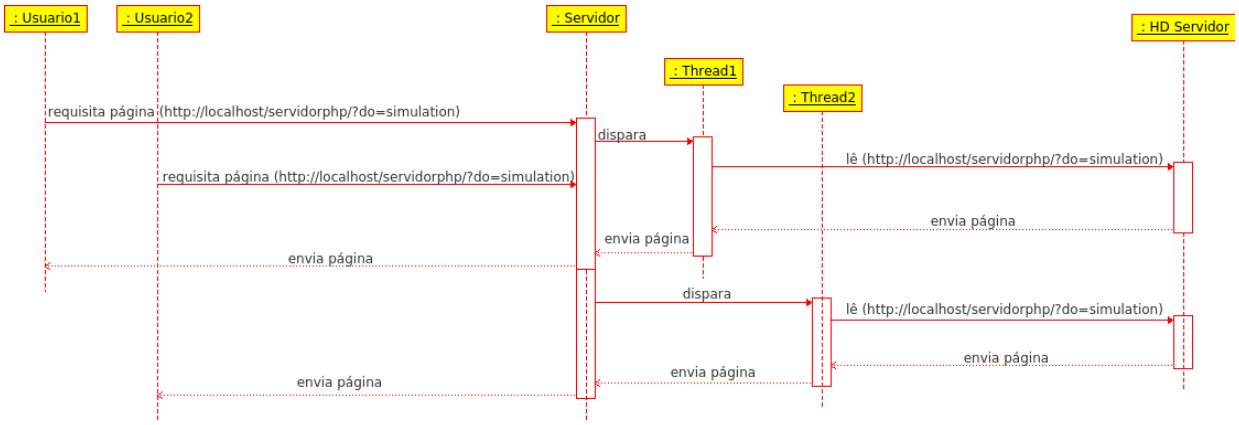

Figura 3.5: Diagrama de Sequência: Servidor MultiThread com um canal de comunicação.

Agora, com as devidas correções, várias solicitações podem ser emparelhadas e disparadas simultaneamente, da seguinte maneira:

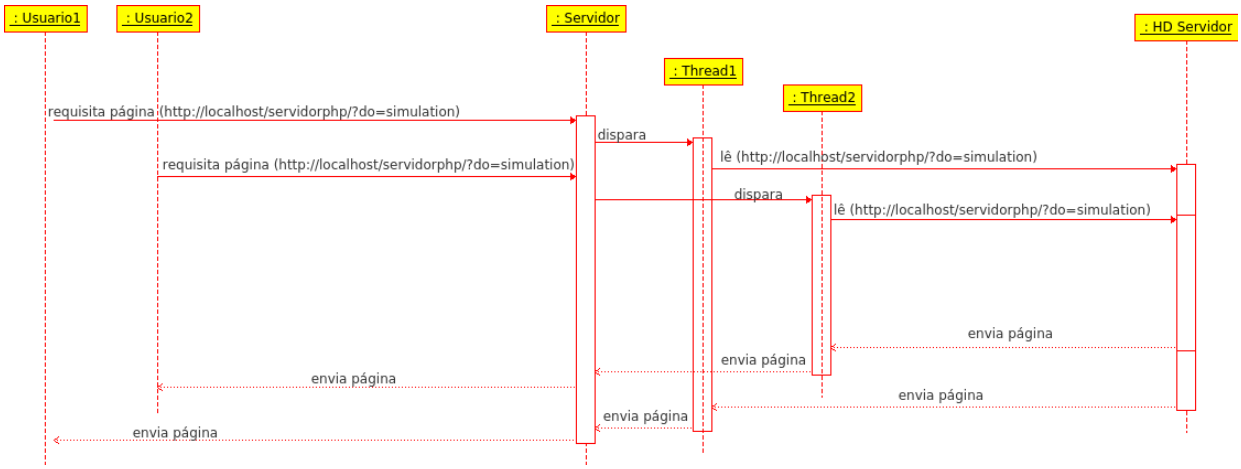

Figura 3.6: Diagrama de Sequência: Servidor MultiThread corrigido.

Portanto, pode-se ver que esta nova funcionalidade foi agregada à ferramenta, visto que anteriormente a este trabalho, o servidor estava funcionando como *singlethread*, criando um "gargalo" no canal de comunicação.

4 Conclusões

O WSDS é um *software* operacional que foi desenvolvido para ser explorado em meio acadêmico e possui uma interface rápida, agradável e intuitiva. Apesar de possuir todas as funcionalidades mínimas necessárias para simular sistemas dinâmicos, ele não foi disponibilizado para uso devido, principalmente, à falta de manutenibilidade do código-fonte, que impedia o acréscimo de novas funções importantes como a análise semântica do compilador. A análise semântica pode ser implementada agora graças à utilização do padrão *Visitor*, que possibilitará ainda que outras funções sejam agregadas ao simulador, devido a sua facilidade de manipulação.

A aplicação do padrão *Bridge*, possibilitou uma melhora significativa na manutenibilidade deste sistema, e também proporcionou que novos métodos de simulação fossem implementados, como o Runge-Kutta 4, que possui maior precisão do que o método de Euler, que era o único a ser executado então.

Além disso, o *Singleton* possibilitará que, de agora em diante, que haja economia de recursos no servidor *web* (que são limitados).

Portanto, tem-se em mente que os objetivos deste trabalho foram alcançados. A RS do simulador utilizou os padrões de projeto como fonte principal de melhoria, e foram exploradas as possibilidades de aplicação dos mesmos.

Como trabalho futuro, espera-se que outros Padrões de Projeto ainda sejam utilizados no WSDS, como por exemplo, o padrão *Flyweight* que possibilitará que estruturas de dados reutilizáveis como a Tabela de Símbolos aumentem o desempenho do *software*.

Referências Bibliográficas

Aho, A. V.; Sethi, R. ; Ullman, J. D. **Compiladores: Princípios, Técnicas e Ferramentas**. , 1995, 351p.

de Oliveira Barros, M. **Gerenciamento de projetos baseado em cenários: Uma abordagem de modelagem dinâmica e simulação**. 2001. 155-156p. Dissertação de Mestrado - COPPE/UFRJ.

de Barros Barbosa, C.; de Almeida, D. D. B. ; da Veiga Silva and, W. **Simulador distribuído de dinâmica de sistemas**. UFJF, 2011. III Workshop de Trabalhos de Graduação e Pós-Graduação - DCC/UFJF.

Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P. ; Stal, M. **Pattern-Oriented Software Architeture: A System Of Patterns**, volume 1. , 1996, 487p.

CHIKOFSKY, E. J.; CROSS II, J. H. Reverse engineering and design recovery: A taxonomy. **IEEE Software**, p. 13–17, 1990.

Coulouris, G.; Dollimore, J. ; Kindberg, T. **Sistemas Distribuídos: Conceitos e Projeto**. , 2007, 43p.

Forrester, J. W. System dynamics and k-12 teachers. **A lecture at the University of Virginia School of Education**, p. 35, May 1996. Massachusetts Institute of Technology Cambridge, MA, USA.

Forio online simulations, 2012.

Gamma, E.; Helm, R.; Johnson, R. ; Vlissides, J. **Padrões de Projeto - Soluções reutilizáveis de software orientado a objetos**. , 1994.

Germoglio, G. M. Arquitetura de software. p. 188, 2009.

de Medeiros Júnior, J. V.; de Oliveira, F. P. S.; de Souza, R. L. R. ; Anez, M. E. M. **Simulação da dinâmica do "jogo da cerveja" através do software de modelagem e simulação empresarial simadm**. p. 10, november 2006. XIII SIMPEP – Bauru, SP,Brasil.

Larman, C. **Applying UML and Patterns - An Introduction to Object-Oriented Analysis and Design and the Unified Process**. , 2002.

Lourden, K. C. **Compiladores: Princípios e Práticas.** 569p.

Powersim software Disponível em <http://www.powersim.com/>. Último acesso: 24/09/2012.

Pressman, R. S. **Engenharia de Software**. Número 8534602379. , 1995, 1056p.

da Silva, M. C. Identificação de estilos de arquiteturas: Um processo dirigido por conhecimento. p. 206, 2000. Pontifícia Universidade Católica do Rio de Janeiro.

Sommerville, I. **Software Engineering**, volume 13. , 2001.

SourceForge.Net. **Quick sequence diagram editor: Multithreading example**. Disponível em <http://sdedit.sourceforge.net/multithreading/example/index.html>. Último acesso: 25/03/2013.

Isee systems. Disponível em <http://www.iseesystems.com/>. Último acesso: 24/09/2012.

Ventana systems, inc. Disponível em <http://vensim.com/>. Último acesso: 23/09/2012.

WARDEN, R. Software Reuse and Reverse Engineering in Practice. 1992, 283-305p.

Wlinkit: Ferramenta de modelagem computacional para educação Disponível em <http://www.nce.ufrj.br/ginape/wlinkit/>. Último acesso: 23/09/2012.

A Exemplo de um modelo de simulação

A.1 Código-fonte de Modelo e Meta-modelo utilizados na simulação da figura 2.4

```
/***** meta-modelo: presa-predador *****/
model ppmm {
  class  pp {
    property ps_natalidade 1;
    property ps_mortalidade 0.1;
    rate (quant_presa) ps_nasc ps_natalidade * quant_presa;
    rate (quant_presa) ps_morte  - quant_predador * ps_mortalidade * quant_presa;
    stock quant_presa 1;

    property pd_natalidade 0.2;
    property pd_mortalidade 0.1;
    rate (quant_predador) pd_nasc pd_natalidade * quant_predador * quant_presa;
    rate (quant_predador) pd_morte  - (pd_mortalidade * quant_predador);
    stock quant_predador 8;
  };
};

```

```
/***** modelo: presa-predador *****/
DEFINE ppm ppmm {
      ps = new pp
      set ps_mortalidade = 0.1;
      set ps_natalidade = 1;
      set pd_mortalidade = 0.1;
      set pd_natalidade = 0.2;
};

```

A.2 Modelo em Dinâmica de Sistemas da figura 2.4

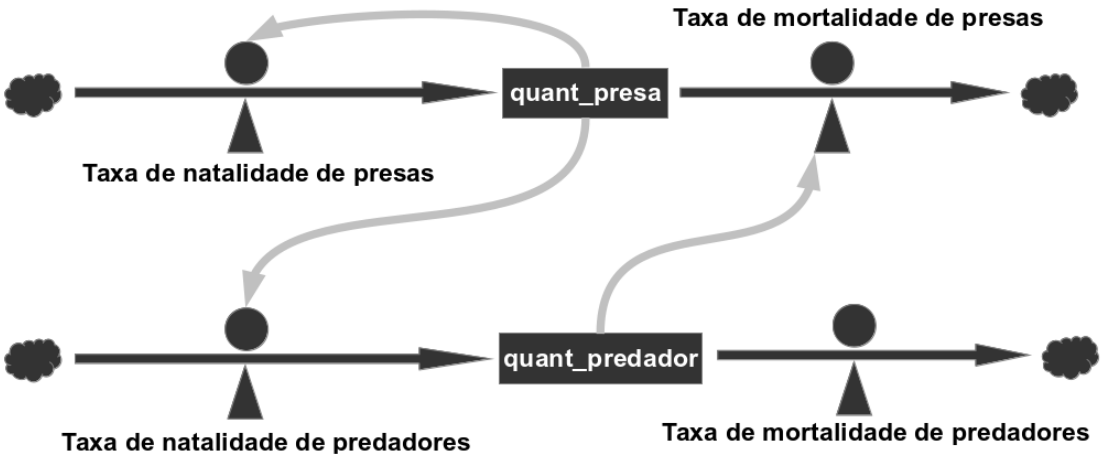

Figura A.1: Modelo presa-predador em Dinâmica de Sistemas.