REUSO NA ANÁLISE DE CUSTO DE CICLO DE VIDA

Denis Gonçalves Cople
Universidade Federal Fluminense
Rua Passo da Pátria 156, sala 440, bloco E, Niterói, 24210-240, RJ.
deniscople@gmail.com

Eduardo Siqueira Brick
Universidade Federal Fluminense
Rua Passo da Pátria 156, sala 440, bloco E, Niterói, 24210-240, RJ.
brick@producao.uff.br

RESUMO:

Sistemas técnicos são ativos compostos por instalações, equipamentos, *softwares* e pessoas, concebidos e desenvolvidos para permitir a implantação de processos produtivos com finalidades específicas (missão do sistema). Com a crescente complexidade desses sistemas na atualidade, torna-se necessária uma pré-avaliação da implantação, operacionalização e desativação dos mesmos, sob a ótica da eficiência e da eficácia. As abordagens que visam avaliar a priori a eficiência de sistemas técnicos são denominadas Análise de Custo de Ciclo de Vida (ACCV), Análise de Custo de Vida Útil (ACVU), ou Análise de Custo Total de Posse (ACTP). Devido à aleatoriedade intrínseca das situações sendo analisadas e à variedade de situações a serem modeladas, simulações computacionais do tipo Monte Carlo são indicadas. Como os problemas ligados à ACCV são de naturezas muito diferentes e os cenários extremamente dinâmicos, torna-se necessário avaliar a melhor forma de construir este ferramental de cálculo, com vistas à sua adaptação às mais diversas situações de utilização e ao reuso. Este trabalho apresenta um arcabouço computacional flexível para efetuar ACVU de sistemas técnicos, dotado de uma biblioteca de ferramentas especialistas reutilizáveis, aliado a técnicas de programação focadas no reuso de código, tais como orientação a objetos, elementos genéricos, anotações e a padronização proporcionada pelos padrões de desenvolvimento no uso de soluções.

PALAVRAS-CHAVE: Engenharia de Sistemas, Simulação, Reuso de Software

ABSTRACT:

Technical systems are assets composed by facilities, equipment, software and man power which are conceived and built to implement productive processes meant to achieve some purpose (system mission). With the growing complexity of modern technical systems it is mandatory to be able to asses in advance the conditions for their implementation, operation and deactivation under the effectiveness and efficiency points of view. The methodologies meant to estimate values for technical systems efficiency indicators are called Life Cycle Cost Analysis (LCCA), Useful Life Cost Analysis (ULCA) or Total Ownership Cost Analysis (TOCA). Due to great amount of calculations involved, to the random characteristic of the problem and to the diversity of situations to be modeled, Monte Carlo simulations is the preferred approach. Due to the randomness and diversity of situations to be modeled it is desirable to use a simulation tool capable of adaptation to these different situations and reuse. This paper presents a flexible LCCA computational framework with a reusable tools library together with programming techniques for reuse, such as object orientation, generic elements, notations and the standardization provided by development standards.

KEY WORDS: System Engineering, Simulation, Software Reuse

Introdução

Sistemas técnicos são ativos compostos por instalações, equipamentos, *softwares* e pessoas, concebidos e desenvolvidos para permitir a implantação de processos produtivos com finalidades específicas (missão do sistema).

Esses sistemas são compostos por uma grande quantidade de ativos físicos (instalações, equipamentos) dedicados à atividade fim e possivelmente distribuídos em várias unidades operacionais em locais geograficamente distintos, além de sistemas capacitadores (estruturas de manutenção e abastecimento, por exemplo), também distribuídos geograficamente.

Quando se projeta um sistema técnico desse tipo para resolver um determinado problema existem duas considerações a serem feitas: a eficácia do sistema e a sua eficiência. A eficácia define o grau de alcance da finalidade do sistema, enquanto a eficiência mede a quantidade de recursos de toda a ordem despendidos para alcançar a eficácia mínima desejável.

Com a crescente complexidade dos sistemas técnicos da atualidade, torna-se necessária uma pré-avaliação da implantação, operacionalização e desativação dos mesmos, sob a ótica da eficiência e da eficácia. Para sistemas complexos, do tipo considerado neste trabalho, o indicador de eficiência mais indicado é chamado de Custo de Ciclo de Vida (CCV). As abordagens que visam avaliar a priori esses indicadores de eficiência são denominadas Análise de Custo de Ciclo de Vida (ACCV). A disponibilidade do sistema, ou a sua capacidade produtiva, é um importante indicador da sua eficácia e a sua avaliação a priori também é desejável e é um subproduto desse tipo de análise.

Essa metodologia de avaliação econômica de projetos deve considerar todos os custos relativos à aquisição, desenvolvimento, operação, manutenção e desativação, desde que os mesmos sejam considerados potencialmente importantes para uma tomada de decisão.

Devido à grande quantidade de cálculos envolvidos, à aleatoriedade intrínseca das situações sendo analisadas e à variedade de situações a serem modeladas, simulações computacionais do tipo Monte Carlo são indicadas. O uso de simulação para resolver problemas desse tipo não é novidade. Entretanto, persistem dois problemas que ainda requerem investigação e novas abordagens: o tempo necessário para realizar simulações envolvendo comparação de uma grande quantidade de alternativas e a variedade de modelos e dados que podem ser usados, dependendo da situação particular de cada organização que procura aplicar o método.

O primeiro é de grande complexidade já que, no limite, se tem um problema combinatório em que a avaliação de cada alternativa é feita com a realização de inúmeras rodadas de um programa de simulação, cada uma delas podendo exigir um grande tempo de computação.

O segundo problema também acrescenta complexidade à solução, já que uma ferramenta computacional, que possa ser usada por diversas instituições, ou por uma mesma instituição em épocas diferentes, para resolver este tipo de problema, tem que ser muito flexível para acomodar modelos e dados disponíveis em cada situação. Isto leva à idéia de construir uma ferramenta de software que seja capaz de aceitar a introdução de novos modelos e processos de cálculo e novos tipos de dados, não contemplados inicialmente na modelagem do problema.

O desenvolvimento de um sistema de simulação expansível para ACCV, com o aproveitamento de toda a capacidade computacional disponível, inclusive através da detecção de pontos onde o paralelismo computacional encontra-se como uma alternativa viável, possibilita o estudo de novos modelos de cálculo, permitindo aos pesquisadores testar estes modelos em condições próximas às reais, com a utilização de dados estocásticos e um número considerável de simulações.

Como os problemas ligados à ACCV são de naturezas muito diferentes e os cenários extremamente dinâmicos, torna-se necessário avaliar a melhor forma de construir este ferramental de cálculo, com vistas para o reuso.

A arquitetura de *framework*, onde é construída uma biblioteca de ferramentas especialistas reutilizáveis pelos programadores, aliado a técnicas de programação focadas no reuso de código, tais como orientação a objetos, elementos genéricos e anotações, e a padronização proporcionada pelos padrões de desenvolvimento no uso de soluções, permite uma utilização facilitada, dinâmica e plenamente adaptável deste ferramental por parte dos desenvolvedores de soluções para ACCV.

O reuso é de extrema importância no desenvolvimento de *softwares*, não só por diminuir o custo e o tempo para a definição de um novo sistema, mas também por aumentar a qualidade com a utilização de componentes que centralizam a manutenção e podem ser mais amplamente testados.

Arquiteturas voltadas para reuso

Robinson et al. (2004) abordaram o problema de reutilização de modelos de simulação e chegaram a conclusões conflitantes. Eles definiram o espectro de reutilização de software como: reutilização do modelo completo, reutilização de componentes, reutilização de funções e clareza do código. Estes tipos de reutilização são ordenados em ordem crescente, de acordo com a freqüência de uso, e por ordem decrescente, de acordo com o critério de complexidade. A conclusão a que chegaram é que, embora a reutilização de modelos seja atraente, supostamente para reduções de custo e tempo de desenvolvimento, há uma série de questões que podem impedir que essas vantagens sejam obtidas. Eles mencionam explicitamente validade e credibilidade do modelo e o custo e tempo para a familiarização, como exemplos de tais questões. Outra conclusão é que a reutilização implica uma arquitetura de *software* que a suporte.

Para Mannisto, Savolainen e Myllarniemi (2008 p.117), as características de um problema complexo podem ser relacionadas com a concepção de uma arquitetura de software, pois não há formulação definitiva, não existe uma regra de parada indicando claramente quando a solução foi encontrada, as soluções não são verdadeiras ou falsas, mas simplesmente boas ou ruins, e finalmente, cada problema pode ser considerado como um sintoma de outro problema. Ainda segundo os autores, arquitetura de software não é uma disciplina isolada, e as decisões são construídas em parte pela adaptação de soluções para problemas encontrados em outros lugares.

Gardazi e Shahid (2009 p.402) alertam que não se deve subestimar a importância do arquiteto de software. Isto se deve ao fato de que uma arquitetura adequada é crucial para a correta implantação de todas as funcionalidades do sistema, além do que a adoção de arquiteturas padronizadas facilita a manutenção e utilização de ferramentas de produtividade.

A organização de um *software* em termos arquiteturais é uma decisão que irá interferir em todas as fases de concepção e desenvolvimento do sistema, definindo os tipos de componentes que poderão ser utilizados, como estes componentes deverão ser integrados, além da forma como este sistema poderá interagir com usuários, recursos externos, serviços e outros sistemas.

A arquitetura *Model-View-Control* (MVC) é uma das mais adotadas na definição de sistemas da atualidade, caracterizando-se por dividir o aplicativo em três camadas:

- **Modelo** (*Model*), referente à camada de persistência;
- **Visualização** (*View*), responsável pela interação com o usuário; e
- **Controle** (*Control*), a qual detém as regras de negócio e faz a intermediação de todas as solicitações efetuadas entre a visualização e o modelo.

Para Dalle, Ribault e Himmelspach (2009 p.947), o padrão MVC pode ser usado na simulação para descrever a dependência entre uma visualização em tempo de execução

(interativa), o modelo executável e o algoritmo de simulação, permitindo, por exemplo, diferentes visualizações simultâneas sobre o modelo a ser criado.

Outro foco de interesse no estudo das formas de reuso são os componentes do tipo *Commercial Off-The-Shelf* (COTS), que tratam de componentes comerciais reutilizáveis, testados e aceitos pelo mercado, e integráveis com relativa facilidade.

Entretanto, mesmo que um COTS apresente meios de integração padronizados, esta integração acrescenta complexidade ao sistema. Para Egyed e Balzer (2006 p. 42) este aumento de complexidade reside no desafio de como viabilizar o uso de COTS que podem efetuar mudanças internas (dados ou estados) quando estas mudanças afetam o sistema como um todo, ou seja, o principal problema de integração é adequar o comportamento do COTS sem alterar o componente em si.

Para Mohamed, Ruhe e Eberlein (2008 p. 147) este problema de integração ocorre porque componentes COTS são produtos comerciais, voltados para uma utilização mais ampla, enquanto requisitos de projeto são específicos para o sistema. Conseqüentemente, os desenvolvedores escolhem quais componentes são mais eficazes para a resolução dos requisitos e, em seguida, tentam resolver o maior número de inadequações mediante adaptação dos selecionados, o que leva a uma metodologia de escolha onde devem ser considerados o custo de adaptação e a eficácia de cada COTS.

Em termos de *software*, a divisão em camadas do MVC permitiu a criação de ferramentas COTS voltadas para a resolução de problemas genéricos em cada um dos três níveis, viabilizando a rápida combinação destas ferramentas no processo de construção do aplicativo, o que diminui o tempo e o custo de desenvolvimento, além de aumentar a qualidade final dos sistemas produzidos.

Atualmente ocorre uma preocupação muito grande acerca da escolha dos COTS adequados, levando em conta fatores como confiabilidade, manutenibilidade, eficiência, eficácia e longevidade do componente. Trabalhos de autores como Jha, Arora, Kapur e Kumar (2010), Gupta, Mehalawat e Verma (2010), Wanyama e Far (2008), demonstram esta preocupação na escolha de COTS, ao definirem metodologias diversas para este fim.

Os vários *frameworks* disponibilizados no mercado de *software*, como o *Java Enterprise Edition* (JEE) e o *Java Persistence Architecture* (JPA), são bons exemplos de componentes do tipo COTS. Cada *framework* resolve apenas um domínio específico, como persistência, controle de serviços transacionais, apresentação visual, etc, e os desenvolvedores precisam saber como combiná-los na concepção de um novo sistema. É importante conhecer também a aceitação do *framework* pelo mercado, pois isto irá definir sua longevidade e manutenibilidade.

Um *framework* é constituído de um grupo de ferramentas genéricas que podem ser utilizadas na resolução de problemas de um domínio específico, podendo ser concebido de diversas formas, mas sendo a abordagem orientada a objetos a preferida pelos desenvolvedores da atualidade.

Frameworks diferem de aplicações comuns, pois dependem de outros aplicativos que os utilizem, complementando as informações necessárias ao seu funcionamento em uma etapa chamada instanciação.

Segundo Horstmann (2007 p. 312), em um *framework* ocorre inversão de controle, o que significa que as classes do *framework* e não as classes da aplicação são responsáveis pelo controle do fluxo da aplicação. O autor também defende a idéia de construir *frameworks* orientados a objetos ao definir um *framework* como um conjunto de classes cooperativas que implantam os mecanismos essenciais para um domínio de problema específico, ou também, um conjunto de classes e interfaces que estruturam o mecanismo essencial de um domínio em particular.

Conforme citado por Cople e Brick (2010 p.12), um programador que utiliza um *framework* não precisa conhecer a estrutura interna, mas apenas saber como requisitar os serviços fornecidos. Em termos de orientação a objetos, *frameworks* são constituídos de

classes abstratas, mas podem utilizar outros tipos de recursos, tais como protocolos de rede ou comunicação direta entre processos.

Segundo Searle e Brennan (2007 p. 3-6), assumindo que o reuso é adequado e viável para a modelagem e simulação, pode-se observar que interoperabilidade e reutilização estão intimamente relacionadas. A fim de reutilizar um determinado componente em um sistema mais amplo, deve-se assegurar que ele possa interagir com os outros elementos do sistema (pelo menos aqueles com os quais deve trocar dados e informações).

Há muito tempo existe a preocupação em reutilizar os sistemas existentes na construção de sistemas maiores e mais complexos através da interoperabilidade, o que levou à definição de arquiteturas e metodologias que permitissem esta integração.

Um exemplo disso foi o surgimento da *High Level Architecture* (HLA) voltada para a integração de módulos de simulação que podem cooperar para um objetivo comum, executando de forma distribuída.

A definição da HLA foi iniciada pelo Departamento de Defesa dos Estados Unidos (DOD) em 1995, sendo a responsabilidade de sua evolução delegada ao *IEEE's Simulation Interoperability Standards Committee* (SISC) no ano de 1997. A partir disto a HLA passou a ser regulamentada pela norma IEEE 1516, *Standard for Modeling and Simulation High Level Architecture - Framework and Rules*, com última revisão ocorrida no ano de 2010.

Segundo Morse et al. (2006 p.115), HLA provê uma metodologia comum para a modelagem de sistemas de simulação distribuídos, conectando simulações e interfaces com sistemas interativos, segundo uma abordagem contemporânea onde o modelo de dados e a semântica da arquitetura são separados das funções e métodos para intercâmbio de informações.

Segundo Papazoglou, Traverso, Dustdar e Leymann (2007 p.64), *Service Oriented Computing* (SOC) é um paradigma que define o uso de serviços para o desenvolvimento de aplicações distribuídas a baixo custo, de forma rápida, interoperável e evolutiva.

Este paradigma é suportado pelas Arquiteturas Orientadas ao Serviço (SOA, do inglês *Service Oriented Architecture*), onde a interface padrão para disponibilização e integração de serviços é implementada através de *Web Services*.

Para Dan, Johnson e Carrato (2008 p.28), a reutilização de serviços em um ambiente SOA proporciona muitos benefícios incluindo **melhorar a agilidade** na implementação de soluções através de uma rápida montagem novos processos de negócios a partir de serviços existentes para atender às mudanças das necessidades de mercado, **reduzir os custos**, não apenas evitando duplicação de código ao reutilizar funções de negócios semelhantes ao longo dos múltiplos processos, mas também durante todo o ciclo de vida do sistema abrangendo a implantação de serviços e gestão, e **reduzir os riscos** através do reuso de código e ambiente de execução bem testados.

Para Fan, Zhangand e Fan (2010 p. 308), a multidisciplinaridade colaborativa das tecnologias de simulação é amplamente utilizada na definição e construção de produtos complexos, sendo a combinação de HLA e *Web Services* um meio efetivo para a obtenção de integração e interoperabilidade nas simulações distribuídas heterogêneas.

Framework de ACCV

A análise do custo do ciclo de vida de sistemas é, na verdade, um conjunto de processos de cálculo ligados à estrutura do sistema, seu desenvolvimento, fabricação, operação, manutenção, gastos com desativação, fatores financeiros, ou qualquer outro elemento de custo abordado durante toda a vida útil do sistema.

Esta metodologia de cálculo leva também a uma definição otimizada da estrutura de apoio logístico, e outros sistemas capacitadores tais como, simuladores para treinamento, unidades fabris, necessários para o sistema principal, considerando tanto fatores geográficos, quanto elementos operacionais e transporte.

Pilla (2003) mapeou a complexidade deste tipo de cálculo, gerando um modelo de entidades lógicas à luz da teoria computacional da orientação a objetos, organizando uma família de classes para suporte ao modelo genérico de cálculo de custo de ciclo de vida. Este modelo tira proveito tanto de elementos de composição, permitindo agrupar cálculos já definidos como em uma linha de produção, como da herança, que permite expandir o modelo ao gerar novos algoritmos de cálculo.

Cople (2004) expandiu os conceitos de Pilla, adequando-os ao desenvolvimento de um *framework* construído com uso da linguagem *Java*, escolhida pelo seu suporte natural ao uso de orientação a objetos e reflexividade.

A evolução destes estudos culminou recentemente com a criação de uma nova versão do *framework* no trabalho efetuado por Cople (2010), onde houve a preocupação de se estabelecer técnicas de reuso simplificadas em termos de codificação de sistemas que utilizem este *framework*, bem como a definição de uma arquitetura baseada em *Web Services*, com o intuito de viabilizar a interoperabilidade do novo produto com as mais diversas tecnologias disponíveis, nos moldes da HLA.

Na versão atual, técnicas de modelagem comportamental, como o uso de elementos genéricos e anotações, se tornaram um grande trunfo na minimização do tempo de desenvolvimento, enquanto a arquitetura de *Web Services* permitiu ao *framework* ficar responsável apenas pelo cálculo e distribuição de processamento, delegando a construção de interfaces e análise de resultados para ferramentas mais especializadas.

Classes genéricas, também chamadas de *templates*, não definem necessariamente uma tecnologia nova em termos de modelagem, sendo utilizada há bastante tempo na linguagem *C++*. No entanto, apenas recentemente linguagens como *Java* e *C#* passaram a utilizar esta técnica em larga escala.

Segundo Rumbaugh, Jacobson e Booch (2000 p.130), *template* é um elemento parametrizado que pode definir uma família de classes ou uma família de funções, apresentando espaços (*slots*) para classes, objetos e valores.

Um elemento genérico, em última análise, acaba por viabilizar a reutilização de algoritmos e estruturas de dados, como no caso de filas, pilhas e árvores, absorvendo para si toda a complexidade do comportamento existente no domínio comum, enquanto delega para as classes que vierem a utilizá-lo apenas as configurações mínimas necessárias.

Anotações, por sua vez, tratam de metadados que podem ser associados a uma classe, método ou propriedade, sem interferir com o código-fonte, porém permitindo o reconhecimento dos mesmos por diversas ferramentas, o que viabiliza o reuso de processos de natureza normalmente complexa, como automatização de persistência, sincronização de processos paralelos, exposição de serviços, entre muitos outros.

Nigul e Mah (2009 p. 417) citam que anotar o código fornece aos programadores diversas vantagens, tais como: a capacidade de combinar metadados úteis com código real, separar as preocupações entre codificação e algum domínio da semântica específica, e permitir a reutilização mais fácil das funcionalidades existentes e do código.

O uso de COTS na construção do *framework* trouxe diversas vantagens, tais como a velocidade de desenvolvimento e a robustez proporcionada pelos mesmos, sendo escolhido o JPA para acesso aos dados, intermediado pelo JEE5, segundo o padrão de desenvolvimento *Session Facade*, deixando o *software* completamente independente da base de dados utilizada, como pode ser observado na Figura 1.

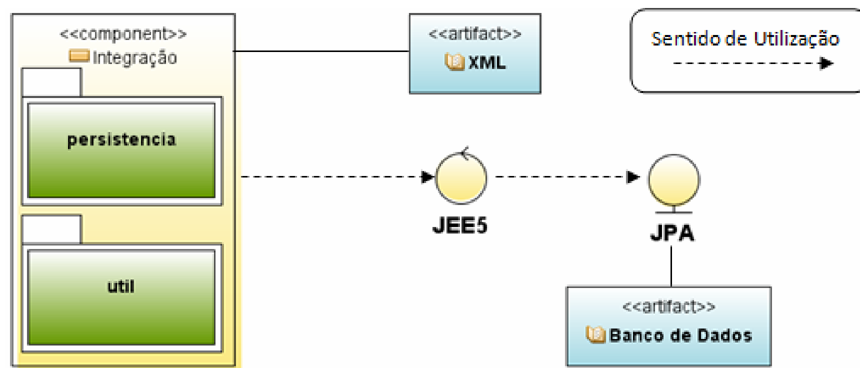

Figura 1: Modelo de persistência do *framework* de ACCV.

O *framework* de ACCV segue o modelo MVC, com intermediação de dados via *Web Services* dentro de uma arquitetura SOA, dividindo as funcionalidades expostas em cinco serviços distintos, disponibilizados para o desenvolvedor:

1. Intermediação de dados cadastrais
2. Construção de Cenários
3. Execução de Simulações
4. Fornecimento de resultados
5. Geração de código-fonte

Apenas a título de ilustração da funcionalidade obtida, o serviço de intermediação de dados cadastrais permite a qualquer desenvolvedor a criação de interfaces visuais para cadastro de dados de catálogo genéricos, ou focados em problemas específicos da empresa onde trabalha, com independência de linguagem, garantindo a interoperabilidade com ferramentas como *Flex*, *Delphi*, *Visual C#*, ou qualquer outra com suporte a *Web Services*. A construção de cenários segue o mesmo princípio, permitindo a criação de modelos de cenários parametrizados específicos para os objetivos de análise de cada empresa.

Os elementos arquiteturais associados aos serviços de intermediação de dados e construção de cenários podem ser observados na Figura 2.

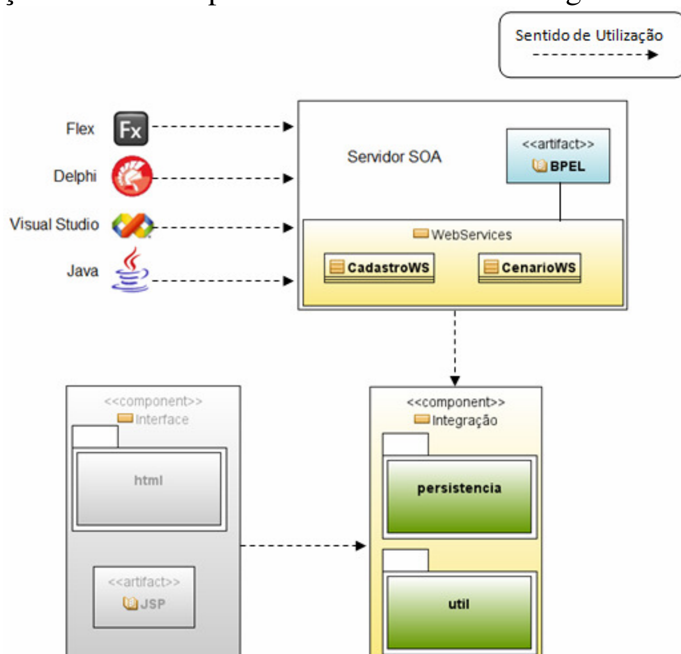

Figura 2: Serviços de Intermediação de Dados e Construção de Cenários

Exemplos de reuso do *Framework* de ACCV

O primeiro exemplo de reuso propiciado pelo *framework* foi a construção de sua própria interface cadastral. Para tal foi utilizado *Flex*, devido à sua acuidade visual, bem como

a possibilidade de se criar telas dinamicamente com o uso de *Action Script*, fator este essencial para a inserção de dados de classes que podem ser extendidas a qualquer momento com o uso de anotações.

Outro fator na escolha do *Flex* como ferramenta de criação para a interface visual foi a facilidade com que o mesmo se integra com os *Web Services*, permitindo que grande parte do desenvolvimento fosse baseado em operações de arraste com geração automática de código.

Apenas para exemplificar os resultados obtidos com esta abordagem, uma das telas cadastrais, referente ao cadastro de órgãos do cenário de simulação é apresentada na Figura 3.

Embora a própria criação da interface cadastral seja um bom exemplo de reuso e interoperabilidade, no trabalho efetuado por Cople (2010) foi gerado também um extrator de resultados para o *Microsoft Excel*, sem a preocupação de se criar um ambiente com janelas, ou qualquer tipo de visualização avançada neste extrator, pois o objetivo era apenas demonstrar a simplicidade do reuso baseado na interoperabilidade a partir das saídas do *Web Service* responsável pelo fornecimento de resultados.

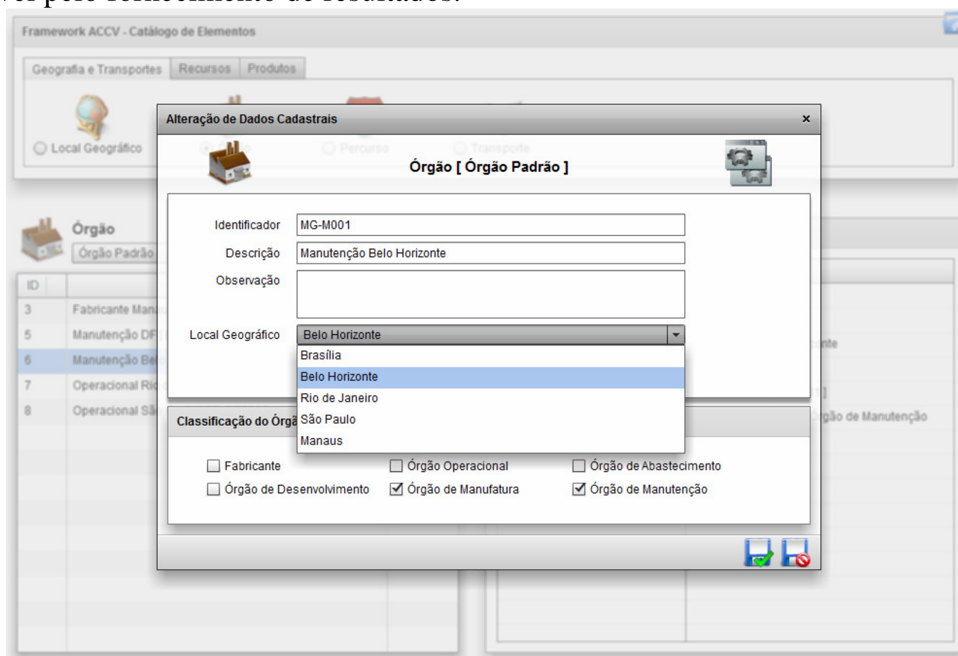

Figura 3: Cadastro de órgãos.

O mapeamento do *Web Service* de fornecimento de resultados do *framework* é reconhecido automaticamente pelo *Visual Studio* a partir da opção “Add Service Reference”, segundo o processo que pode ser visualizado na Figura 4.

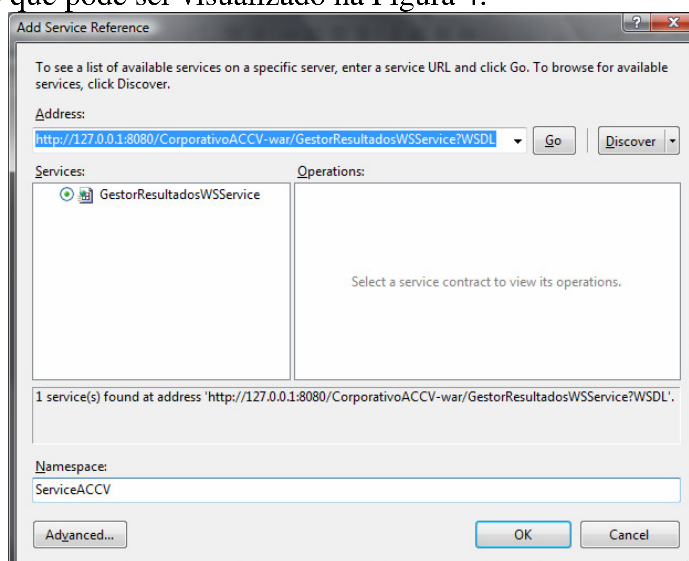

Figura 4: Adição do *Web Service Java* ao sistema em *C#*.

Após a confirmação, as classes *C#* responsáveis pelo encapsulamento das chamadas são criadas, sendo reconhecidas automaticamente pelo ambiente de desenvolvimento, inclusive com a opção de completar código, como pode ser visto na Figura 5.

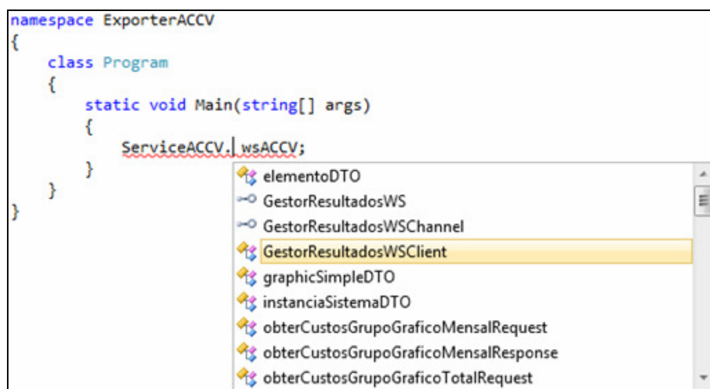

Figura 5: Reconnhecimento do *Web Service* pelo *C#*.

Um processo similar é efetuado com o componente de integração com *Excel*, desta vez utilizando tecnologia *Active X*, amplamente utilizada para viabilizar a interoperabilidade entre ferramentas com tecnologia *Microsoft*.

Com pouco esforço para a criação de uma ferramenta extratora, foi demonstrada como a interoperabilidade facilitou o reuso do *framework* de ACCV em ambientes heterogêneos.

A mesma metodologia poderia ser aplicada para a construção de uma camada de integração com HLA, embora a complexidade do aplicativo a ser gerado possa ser um pouco maior, devido à formalização inerente às normas.

Com o extrator pronto, ele foi executado a partir da linha de comando e os dados puderam ser manipulados diretamente no Excel, como pode ser observado na Figura 6.

Em termos de reuso de classes, as anotações acabam por definir justamente os pontos de flexibilização do novo *framework*, já que qualquer classe cujos metadados devam ser armazenados no banco de dados deverá utilizá-las, permitindo o processamento e posterior exportação destes metadados para a interface cadastral padrão ou qualquer outra que venha a ser criada.

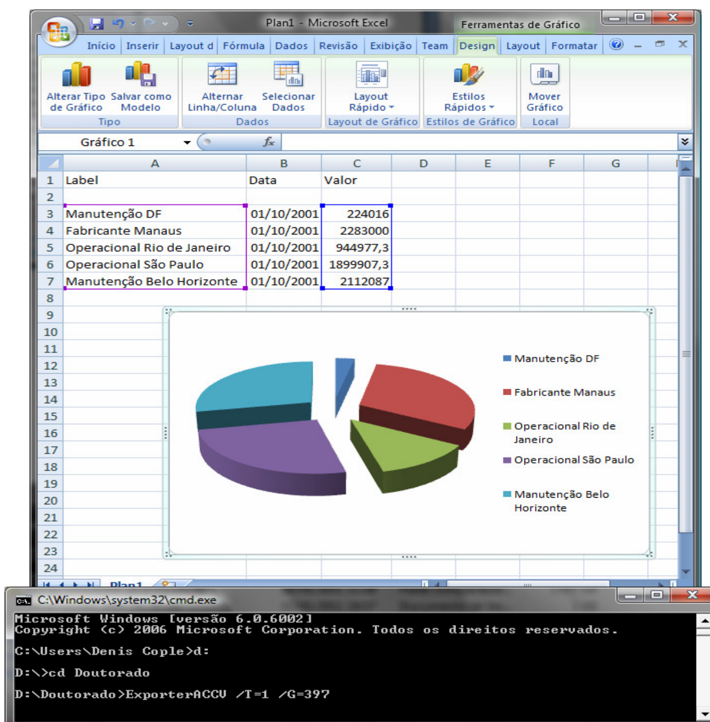

Figura 6: Resultado da extração de resultados para o *Microsoft Excel*.

A Figura 7 apresenta um novo tipo de atividade, no caso “Transporte com Seguro”, sendo oferecida pela interface cadastral.

Figura 7: Exibição de novo tipo de atividade na interface cadastral.

Conclusões

Não é interessante desenvolver uma solução tão genérica quando o foco é a resolução de um problema limitado, porém o uso do *framework* para diferentes contextos de ACCV exige adaptações que apenas serão possíveis devido às características de reuso apresentadas.

O uso de anotações para a extensão do *framework* trouxe a tarefa a um nível muito mais formal dentro do ambiente de programação. Mesmo não sendo necessário o uso de XML, especulou-se inicialmente sua adoção em paralelo, mas a idéia foi abandonada já que se tornou um caminho redundante e pouco eficiente.

Futuramente espera-se implantar este encapsulamento em XML com objetivos mais audaciosos. Uma das frentes de desenvolvimento que deverá continuar a partir do término deste trabalho será a construção de uma ferramenta para definição visual de classes e máquinas de estado compatíveis com o *framework*, de forma a levar sua utilização até as pessoas totalmente leigas em termos de programação.

A arquitetura orientada a serviços foi utilizada de forma conceitual, inclusive hospedando os *Web Services*, em um servidor SOA (*GlassFish*), mas não houve necessidade de se utilizar BPEL. Isto não impede que novos pesquisadores combinem estes serviços de forma a gerar soluções totalmente inovadoras em trabalhos futuros, soluções estas que sequer podem ser vislumbradas aqui.

Com a plena interoperabilidade obtida com o uso de *Web Services* foi possível aproveitar as melhores características da linguagem *Java* em termos de processamento e distribuição, ao mesmo tempo em que telas muito complexas e com grande requinte visual foram construídas através do *Flex*. Novos sistemas poderão ser construídos em ambientes como *Visual Studio .Net*, PHP, ou qualquer outra linguagem, devido às características de comunicação e exposição de interface dos *Web Services*.

Isso também deixa o simulador alinhado com as tendências atuais para a criação de simulações distribuídas, já que o modelo de utilização do *framework* é baseado em serviços, com exposição através de um padrão amplamente utilizado, o que viabiliza a integração com outros simuladores e ferramentas de interesse em ambiente HLA.

Referências

COPLE, D. G. Simulação do Custo de Ciclo de Vida de Sistemas Técnicos. Brasil: UFF, Dissertação de Mestrado, 2004.

COPLE, D. G. Um Framework para Análise de Custo de Ciclo de Vida baseado em Reuso e Interoperabilidade. Brasil: UFF, Tese de Doutorado, 2010.

COPLE, D. G; BRICK, E. S. A simulation framework for technical systems life cycle cost analysis. USA: Elsevier, Simulation Modelling Practice and Theory 18, 2010, p. 9-34.

DALLE, O; RIBAULT, J; HIMMELSPACH, J. Design Considerations for M&S Software. USA: IEEE, Proceedings of the 2009 Winter Simulation Conference, 2009, p. 944-955.

DAN, A; JOHNSON, R. D; CARRATO, T. SOA Service Reuse by Design. USA: ACM, SDSOA'08 Proceedings of the 2nd international workshop on Systems development in SOA environments, 2008, p. 25-28.

EGYED, A; BALZER, R. Integrating COTS Software into Systems through Instrumentation and Reasoning. Holanda: Springer Science and Business Media, Inc, Automated Software Engineering, Vol. 13, 2006, p. 41-64.

FAN, M; ZHANGAND, J; FAN, Y. A heterogeneous model integration and interoperation approach in distributed collaborative simulation environment. UK: Inderscience Publishers, International Journal of Internet Manufacturing and Services, Vol. 2, 2010, p. 294-309.

GARDAZI, S. U; SHAHID, A. A. Survey of software architecture description and usage in software industry of Pakistan. Paquistão: Islamabad, ICET 2009, Emerging Technologies - International Conference, 2009, p. 395-402.

GUPTA, P; MEHLAWAT, M. K; VERMA, S. COTS selection using fuzzy interactive approach. Alemanha: Springer Berlin, Optimization Letters, 2010, p. 1-17.

HORSTMANN, C. Padrões e Projeto Orientados a Objetos - Segunda Edição, versão traduzida. Brasil: Bookman, 2007, 423 p.

JHA, P. C; ARORA, R; KAPUR, P. K; KUMAR, U. D. Optimal component selection of COTS based software system under recovery block scheme incorporating execution time. India: Springer India, International Journal of Systems Assurance Engineering andManagement, Vol. 1, No. 1, 2010, p. 77-83.

MANNISTO, T; SAVOLAINEN, J; MYLLARNIEMI, V. Teaching Software Architecture Design. Canada: Working IEEE/IFIP Conference on Software Architecture (WICSA 08), 2008, p. 117-124.

MOHAMED, A; RUHE, G; EBERLEIN. A. Sensitivity analysis in the process of COTS mismatch-handling. Alemanha: Springer-Verlag, Requirements Eng 13, 2008, p. 147-165.

MORSE, K. L; LIGHTNER, M; LITTLE, R; LUTZ, B; SCRUDDER, R. Enabling Simulation Interoperability. USA: Computer, Vol. 39, No. 1, 2006, p. 115-117.

NIGUL, L; MAH, E. Software maintainability benefits from annotation-driven code. USA: IEEE International Conference on Software Maintenance, 2009, p. 417-421.

PAPAZOGLOU, M. P; TRAVERSO, P; DUSTDAR, S; LEYMANN, F. Service-Oriented Computing: State of the Art and Research Challenges. USA: IEEE Computer Society, Vol. 40, No. 11, 2007, p. 38-45.

PILLA, L. H. Sistemas de Apoio à Decisão: uma Contribuição ao Estudo de Análise de Custo de Ciclo de Vida. Brasil: UFF, Dissertação de Mestrado, 2003.

ROBINSON, S; NANCE, R. E; PAUL, R. J; PIDD, M; TAYLOR, S. J. E. Simulation model reuse: definitions, benefits and obstacles. USA: Elsevier, Simulation Practice and Theory 12, 2004, p. 479-494.

RUMBAUGH, J; JACOBSON, I; BOOCH, G. UML: Guia do Usuário, 3a ed, versão traduzida. Rio de Janeiro: Editora Campus, 2000, 472 p.

SEARLE, J; BRENNAN, J. General Interoperability Concepts. França: RTO, Report apresentado em conferência da NATO, Integration of Modelling and Simulation, Educational Notes RTO-EN-MSG-067, Paper 3, 2007, p. 3.1-3.8.

WANYAMA, T; FAR, B. H. An Empirical Study to Compare Three Methods for Selecting Cots Software Components. Uganda: International Journal of Computing and ICT Research, Vol. 2, No. 1, 2008, p. 34-46.