Um Arcabouço *open source* em Python para DBC com Suporte à Evolução Dinâmica não Antecipada

Yguaratã C. Cavacanti[1], **Hyggo Oliveira de Almeida**[1], **Evandro Costa**[2]

[1]Instituto de Computação – Universidade Federal de Alagoas
Maceió – AL – Brasil

[2]Departamento de Sistemas e Computação – Universidade Federal de Campina Grande
Campina Grande – PB – Brasil

yguarata@gmail.com, evandro@tci.ufal.br, hyggo@dsc.ufcg.edu.br

Abstract. *This document presents a framework in Python language that enable the software development using the CBD(Component-based Development) methodology with unanticipated dynamic software evolution support.*

Resumo. *Este trabalho apresenta um arcabouço desenvolvido em linguagem Python que possibilita o desenvolvimento de software utilizando a metodologia DBC(Desenvolvimento Baseado em Componentes) com suporte à evolução dinâmica não antecipada de software.*

1. Introdução

Um dos grandes problemas na Engenharia de Software, mais especificamente com relação à evolução do software, são as mudanças de requisitos não previstas que podem ter impacto direto sobre o projeto e o código do software. Essas modificações não podem ser antecipadas durante a fase de especificação, e por esse motivo estão sendo apontadas como as principais causas de problemas técnicos e do aumento do custo do projeto em si.

Apesar de recentes avanços na Engenharia de Software terem introduzido novos conceitos e tecnologias que promovem uma maior flexibilidade e agilidade no desenvolvimento de aplicações, como arcabouços de software [Biggerstaff and Richter 1987], sistemas baseados em componentes [Kozaczynski and Booch 1998], arquitetura voltada para serviços [Bichler and Lin 2006] e desenvolvimento baseado em *plugins* [Mayer et al. 2003], os mesmos ainda não são capazes de suprir essas mudanças não antecipadas de uma forma dinâmica. Ou seja, todas essas tecnologias ainda não são aptas a evoluir um sistema sem que o mesmo tenha sua execução interrompida.

É com base neste contexto que é apresentada, neste artigo, a implementação em Python de um arcabouço *open source* de software, denominado PyCF(Python Component Framework), o qual dá suporte à evolução dinâmica de aplicações. A construção de aplicações utilizando o PyCF é feita através da abordagem de DBC(Desenvolvimento Baseado em Componentes), possibilitando, dessa forma, o desenvolvimento rápido e menos complexo das mesmas. A implementação do PyCF é baseada na Especificação de Modelo de Componentes COMPOR, disponível sobre a licença GPL e descrita por [Almeida et al. 2006].

A linguagem Python foi escolhida para tal finalidade por ser uma linguagem de sintaxe simples, com curva de aprendizado baixa e que possibilita a prototipação rápida

de aplicações. Essa última característica é importante, no contexto do PyCF, porque os componentes podem ser construídos através da utilização de protótipos. Além dessas características, Python também favorece a evolução dinâmica de aplicações por ser uma linguagem dinâmica e interativa.

2. CMS: COMPOR Component Model Specification

A CMS é a especificação de modelo de componentes para a construção de software com suporte à evolução dinâmica não antecipada. Na CMS são estabelecidos mecanismos que tornam possível a mudança de qualquer parte do software, removendo e adicionando componentes, mesmo em tempo de execução. Essa especificação também define alguns conceitos como Componentes Funcionais, os quais são estruturas atômicas de software que implementam alguma funcionalidade; e Contêineres, que são utilizados como interface de acesso a outros componentes.

As definições presentes na CMS dizem como devem funcionar o modelo de disponibilização de componentes, o qual é baseado em composição hierárquica, e os modelos de interação baseada em serviços e eventos. Também define o funcionamento dos mecanismos de adaptação de componentes e sobreposição de serviços.

2.1. Modelo de Disponibilização de Componentes

No modelo de disponibilização, os componentes são inseridos dentro de contêineres, os quais também podem conter outros contêineres. Cada contêiner possui tabelas contendo os serviços e eventos de seus componentes filhos, e quando algum componente é inserido ou removido essas tabelas são atualizadas. Os contêineres mantêm essas tabelas porque as funcionalidades dos componentes são acessadas através de serviços e eventos, ou seja, um componente não interage diretamente com outro componente, ele apenas solicita a execução de um serviço ou envia/recebe uma notificação de evento através da hierarquia. Esse mecanismo faz com que os componentes sejam inseridos e removidos sem alterar a estrutura da hierarquia.

2.2. Modelo de Interação Baseada em Serviços e Eventos

Como mencionado anteriormente, os componentes disponibilizam suas funcionalidades por meio de serviços e eventos, fazendo com que não haja referência explícita entre componentes. Em outras palavras, quando um componente deseja alguma funcionalidade ele faz uma solicitação de serviço para a hierarquia da aplicação. Então, se algum componente implementar esse serviço, a solicitação é recebida e o serviço é executado por parte do componente implementador do serviço.

O mesmo acontece com os eventos, ou seja, os componentes anunciam notificações de eventos para a hierarquia da aplicação, e se algum componente estiver interessado nesse evento o mesmo receberá a notificação de evento. Como dito anteriormente, os serviços e eventos são encaminhados para os componentes de acordo com as tabelas mantidas pelos contêineres.

2.3. Adaptando Serviços e Eventos

Uma das principais características da abordagem de DBC é a reutilização de componentes de software pré-existentes. E é com base nessa afirmação que a CMS estabelece

um mecanismo de adaptação de serviços e eventos de componentes. A adaptação de serviços e eventos é uma forma de reutilizar componentes pré-existentes, os quais implementam serviços e eventos desejados nas aplicações, porém com nomes(de serviços e eventos) diferentes. A adaptação desses componentes permite que os nomes dos serviços e eventos do mesmo sejam adaptados de forma a satisfazer as exigências dos componentes da aplicação em questão. Em outras palavras, é possível adaptar a interface de um componente à interface exigida pelos componentes de outra aplicação.

2.4. Sobreposição de Serviços

Os mecanismos de interação e disponibilização de componentes da CMS permitem que serviços sejam sobrepostos mesmo em tempo de execução. Ou seja, assim como no conceito de sobreposição de métodos em programação orientada a objetos, é possível, através de mecanismos definidos pela CMS, sobrepor algum serviço disponibilizado por um componente por um outro serviço de mesmo nome porém com implementação diferente. Dessa forma, quando um componente é disponibilizado e o mesmo possui serviços com nomes iguais aos serviços oferecidos por outros componentes já existentes na aplicação, os serviços já existentes serão sobrepostos pelos serviços do novo componente.

3. Proposta do Artigo: PyCF - Python Component Framework

O PyCF é um arcabouço *open source* de software, construído em Python, que implementa os mecanismos definidos na CMS. O PyCF permite que aplicações sejam construídas utilizando a abordagem de DBC e oferece suporte à evolução dinâmica de aplicações. Nas sub-sessões seguintes serão demonstradas as implementações das principais características do PyCF.

3.1. Arquitetura do PyCF

A base da implementação do PyCF é o padrão de projeto *Composite* [Gamma et al. 1995], o qual possibilita a construção de estruturas hierárquicas necessárias ao funcionamento dos mecanismos definidos na CMS. A Figura 1 mostra um diagrama simplificado da arquitetura básica do PyCF. Os métodos declarados em *AbstractComponent* são implementados de maneira diferente nas classes *Conteiner* e *FunctionalComponent*, tanto para o modelo de interação de serviços como de eventos. Basicamente, a implementação dos métodos em *FunctionalComponent* são voltados para um objeto em si, enquanto em *Container* os métodos são aplicados a todos os objetos da composição.

3.2. Modelos de Interação

No PyCF as funcionalidades oferecidas pelos componentes são disponibilizadas como serviços e eventos, assim como definido na CMS. A seguir, apresenta-se a implementação dos modelos de interação baseados em serviços e eventos.

3.2.1. Implementação do Modelo de Interação Baseado em Serviços

O modelo de interação baseado em serviços é realizado através dos métodos *doIt(...)* e *receiveRequest(...)*, os quais são executados iterativamente. O resultado da interação entre esses dois métodos é um objeto do tipo *ServiceResponse*, o qual encapsula o resultado da

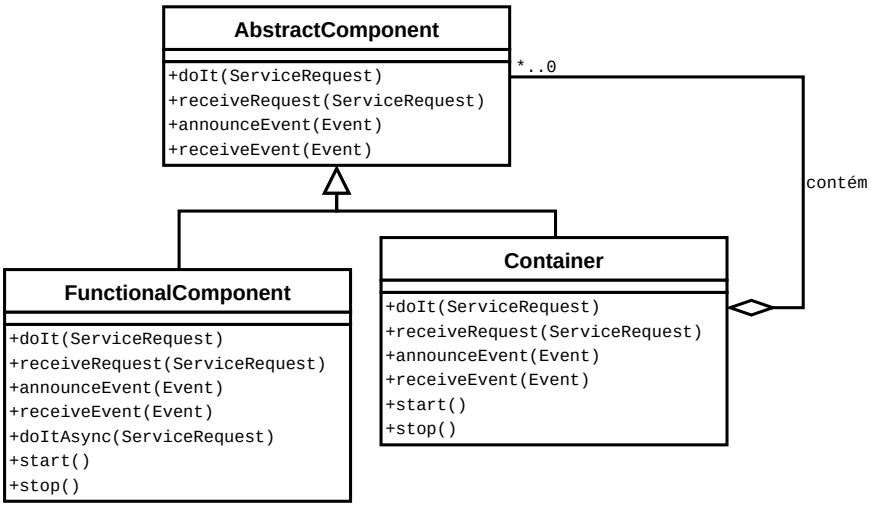

Figure 1. Diagrama de classes básico do PyCF.

execução do serviço ou a exceção gerada, se a mesma ocorrer. A Figura 2 ilustra esse processo.

De acordo com a Figura 2, o componente X faz uma requisição ao serviço *save* através do método *doIt(...)* e esse último encaminha a requisição para a hierarquia da aplicação. A execução do método *doIt(...)* é feita de baixo para cima, então o próximo componente da hierarquia a receber a requisição é o *Container 2*. Como esse último não possui nenhum componente que implemente esse serviço, então o mesmo repassa a requisição para o próximo componente, *Container 1*. O *Container 1* vê em sua tabela de serviços que o *Container 3* possui um componente que implementa o serviço, então o mesmo executa o método *receiveRequest(...)* para a requisição trazida pela método *doIt(...)*. Como a iteração do método *receiveRequest(...)* é feita de cima para baixo, então o próximo componente a receber a requisição é o *Container 3*, o qual através do mesmo procedimento repassa a requisição para o componente K, que é o implementador do serviço *save*. O componente K executa o serviço e retorna um objeto do tipo *ServiceResponse* contendo o resultado da execução ou a exceção gerada.

O PyCF também dispõe de uma implementação assíncrona do modelo de interação baseado em serviços, que é realizada pela interação dos método *doItAsync(...)* e *receiveAsyncServiceResponse*. Esse último método não possui uma implementação padrão, portanto o mesmo deve ser implementado pelo desenvolvedor do componente.

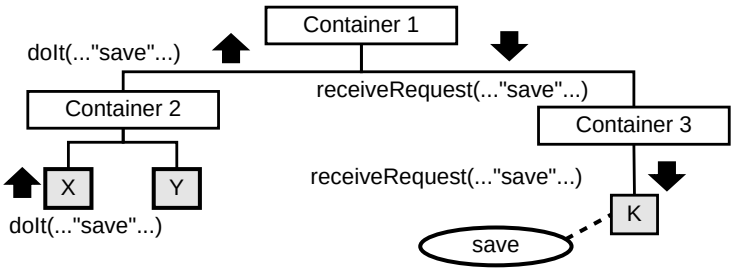

Figure 2. Execução dos métodos *doIt* e *receiveRequest* [Almeida et al. 2006].

3.2.2. Implementação do Modelo de Interação Baseado em Eventos

O modelo de interação baseado em eventos é implementado através da interação entre os métodos *announceEvent(...)* e *receiveEvent(...)*. A interação desses dois métodos é semelhante ao modelo de interação baseado em serviços, com a exceção de que os anúncios de eventos são feitos de maneira assíncrona e não retornam nenhum objeto para o componente anunciante.

3.3. Inicialização e Finalização de Componentes

O PyCF possui um mecanismo de inicialização de propriedades, as quais são armazenadas em uma tabela para cada componente. Essas propriedades podem ser inicializadas através do método *putInicializationProperty(...)*, e podem ser acessadas pelo método *getInicializationProperty(...)*. O PyCF também fornece uma maneira de iniciar e parar a execução dos componentes através dos métodos *start()* e *stop()*. No contexto dos contêineres, os métodos *start()* e *stop()* inicializam e interrompem a execução de todos os componentes do contêiner em questão, respectivamente.

A inicialização e finalização, mencionadas anteriormente, também podem ser personalizadas sobrescrevendo os métodos *startImpl()* e *stopImpl* nos componentes funcionais. Essa característica é importante porque às vezes se faz necessário alguma configuração extra nos componentes durante a inicialização e finalização dos mesmos.

3.4. Scripts de Execução

No PyCF é definida uma classe chamada *ExecutionScript* cuja finalidade é fazer chamadas iniciais a serviços fornecidos por componentes. Esse mecanismo é útil pois, desde que aplicações podem ser construídas juntando-se componentes pré-existentes, conforme a abordagem de DBC, as configurações iniciais de cada aplicação são únicas. Como por exemplo, chamadas a serviços de interfaces gráficas e identificação de usuários.

Na Figura 3 pode-se observar um diagrama simplificado de uma aplicação baseada em componentes construída com o PyCF. Note que no topo da hierarquia existe um objeto do tipo *ScriptContainer*, o qual controla as requisições de um objeto do tipo *ExecutionScript* às funcionalidades de outros componentes da hierarquia. O objeto do tipo *ScriptContainer* deve ser a raiz da hierarquia, para que o *script* de execução tenha acesso a todos os serviços e eventos da aplicação.

A justificativa para se construir *scrips* de execução no lugar de componentes funcionais é que esses últimos dificilmente seriam reutilizados, já que possuem apenas chamadas a serviços iniciais de acordo com as características de cada aplicação.

4. Conclusão

O desenvolvimento do PyCF é uma contribuição significativa para a comunidade de desenvolvedores Python, já que atualmente não existe nenhum arcabouço, inclusive de caráter *open source*, que forneça as características e os objetivos da CMS. Como implementação da CMS, o PyCF torna o desenvolvimento das aplicações mais rápido e menos complexo através da abordagem de DBC, além de dar suporte à evolução dinâmica não antecipada das mesmas.

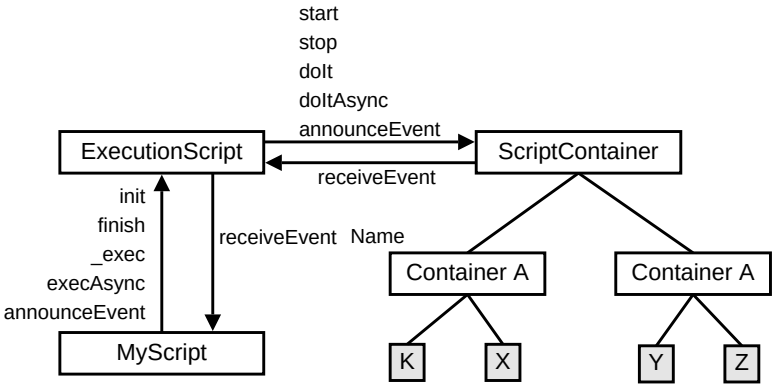

Figure 3. Scripts de execução [Almeida et al. 2006].

Como trabalho futuro sugere-se o desenvolvimento de um servidor de aplicações baseadas em componentes (Python Component Application Server - PyCAS), o qual fornecerá os recursos necessários às atividades de gerenciamento de ciclo de vida e integração dos componentes.

References

Almeida, H., Perkusich, A., Ferreira, G., Loureiro, E., and Costa, E. (2006). A Component Model to Support Dynamic Unanticipated Software Evolution. In *Eighteenth International Conference on Software Engineering and Knowledge Engineering (SEKE'06)*, pages 261 – 267, San Francisco, USA.

Bichler, M. and Lin, K.-J. (2006). Service-oriented computing. *Computer*, 39(3):99–101.

Biggerstaff, T. J. and Richter, C. (1987). Reusability framework, assessment, and directions. *IEEE Software*, 4(2):41–49.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley.

Kozaczynski, W. and Booch, G. (1998). Component-Based Software Engineering. *IEEE Software*, 15(5):34–36.

Mayer, J., Melzer, I., and Schweiggert, F. (2003). Lightweight plug-in-based application development. In *NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 87–102, London, UK. Springer-Verlag.