

A Component Model to Support Dynamic Unanticipated Software Evolution

Hyggo Almeida, Angelo Perkusich, Glauber Ferreira, Emerson Loureiro, and Evandro Costa

Embedded Systems and Pervasive Computing Lab,

Center of Electrical Engineering and Informatics, Federal University of Campina Grande

Postal Code 10.105 - 58109-970, Campina Grande, PB, Brazil

{hyggo, perkusic, glauber, emerson}@dee.ufcg.edu.br

Abstract

This paper presents a component model to support dynamic unanticipated software evolution. Such a model provides mechanisms for managing unpredicted software changes on the fly. A Java implementation of the proposed model is also presented. Finally, an application of the proposed infrastructure in the context of pervasive computing is described.

1. Introduction

Software evolution is a set of activities that occurs after the initial delivery of the software and typically deals with bugfixes and the addition, change or removal of functionalities. When the evolution cannot be anticipated and the software is not prepared for changes, the impact over the existing design and code increases [4]. These unanticipated changes have been pointed as the main cause for most technical problems and related costs of software evolution [8].

Recent advances in Software Engineering, such as application frameworks [5], component based systems [3], service oriented architectures [10], and plugin based development [9], have not been conceived to support unanticipated changes. In fact, some approaches provide flexibility for changes, even at runtime. But, this is only valid for specific parts of the software, which are predicted to change - framework hot spots, services and plug-in interfaces, etc.

The greatest problem arises when part of the software which was thought to be fixed needs to be changed. The unanticipated changes force developers to extensively modify the existing software architecture, design, and code. A change is considered “unanticipated” if its implementation does not depend on hooks encoded in previous versions of the changed software [8].

By definition, “unanticipated software evolution is not something for which we can prepare during the design of a software system”¹. Therefore, a support for unanticipated software evolution must not require developers to specify which part of the software could evolve. Any part of the

software must inherently support evolution, and developers should not bother themselves about the mechanisms that allow this evolution. Moreover, in the case of systems with frequent requirement changes or when the execution of such systems cannot be interrupted, evolution must be managed at runtime.

In this paper we propose a component model to support dynamic unanticipated software evolution, named COMPOR COMPONENT MODEL SPECIFICATION (CMS). Such a model allows to change any part of the constructed software, removing and adding components, even at runtime. The CMS promotes unanticipated software evolution through a simple and lightweight design model. Component based concepts, such as containers and components, are used to build applications based on a hierarchical composition. We present a Java implementation of the CMS, called JAVA COMPONENT FRAMEWORK (JCF). By using the JCF, it is possible to develop Java applications that can be changed during runtime, even for unpredicted changes. Finally, an application of the proposed infrastructure for developing a pervasive computing middleware is presented.

The remainder of this paper is organized as follows. Section 2 describes the CMS. Section 3 introduces the JCF. An application of our approach is presented in Section 4. Related works are discussed in Section 5. Finally, the concluding remarks are presented in Section 6.

2. Component Model Specification

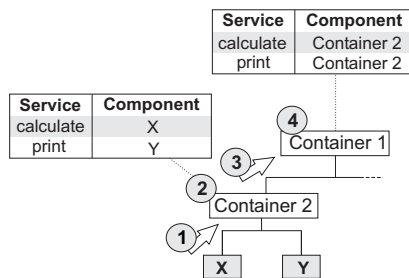
According to the COMPOR COMPONENT MODEL SPECIFICATION (CMS), a component based system must be described as a composition of two kinds of entities: functional components and containers. Functional components are software entities that implement the application-specific functionalities, making them available by means of services and events. A functional component is not composed of other components, that is, it has no child components. Functional components represent the atomic architectural elements of the software application according to the CMS.

Containers are software entities that implement no application-specific functionalities. A container manages the access to the services and events provided by its child components, which may be functional components or other

¹FUSE Workshop - ETAPS 2004

containers. Functional components are made available by inserting them into containers. It is necessary to have at least one container, named *root container*, in order to insert functional components and run the application. Each container has a list of provided services as well as a list of events of interest for its child components.

As mentioned before, each container has a list of provided services as well as a list of events of interest for its child components. Therefore, after inserting a new component in a given container, the list of services and events for each container up to the root of the hierarchy must be updated accordingly. This is necessary in order to make available the services provided by the inserted component as well as to allow the notification of the occurrence of events that are of interest of the component. Figure 1 depicts the component deployment process and its steps are detailed as follows.



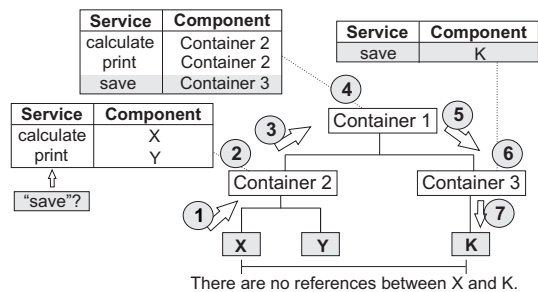
1. A component *X* implementing the “calculate” service is added to the *Container 2*.
2. *Container 2* updates the service table adding the services provided by the component *X* (the same occurs for the event table).
3. *Container 2* asks its parent container, *Container 1* in this case, to update its service table, which adds the services provided by the *Container 2*.
4. *Container 1* updates its service table.

After the execution of these steps the services provided by the component *X* can be accessed from any component in the hierarchy without an explicit reference to it. Note that a component has only the reference to its parent container. In the case of two or more components having services or events with the same identifier, an *alias* is used. Thus, it is possible to register a nickname for each service, allowing services providers to coexist within the same application and be diversified in terms of non-functional features. The remove operation is similar to the deployment, but after removing a component the next invocations for its services will not work.

The interaction model is based on services and events. In the first case any component may invoke a service of another component, even when the component belongs to another container. The interaction based on events focuses on the announcement of a state change in a given component to all the interested components. In both cases there is no explicit reference among components.

2.2.1 Service based interaction

As said before, after inserting a component in a given container, its services are made available to any other components in the application. Therefore, assuming the existence of a service named “save” implemented by a component *K* of the application, the execution of this service can be requested by a component *X*, without an explicit reference to *K*. In Figure 2 such an interaction process is illustrated and detailed as follows.



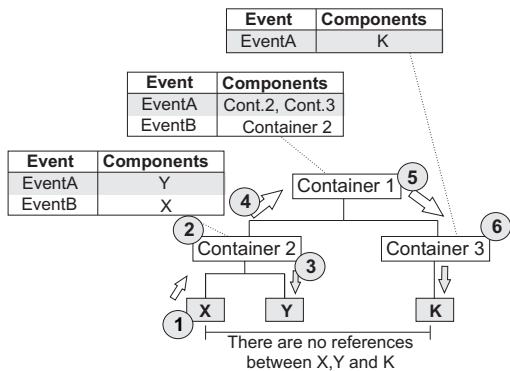
1. Component *X* requests the execution of the “save” service to its parent container.
2. Based on its service table, *Container 2* verifies that no child component implements the “save” service.
3. *Container 2* forwards the request to its parent container, in this case *Container 1*.
4. *Container 1*, according to its service table, verifies that one of its children implements the “save” service, *Container 3* in this case. The *Container 1* sees *Container 3* as the component that implements the requested service.
5. *Container 1* then forwards the service request to the *Container 3*.
6. *Container 3* does not implement the service but has a reference to the component that implements the requested service, and forwards the request to it, in this case component *K*.
7. Component *K* executes the “save” service and returns the result following the reverse path, back to the requester.

It is important to point out that there are no references between the component requesting the service (X) and the component that provides it (K). Thus, it is possible to change the component that provides the “save” service without modifying the rest of the structure.

2.2.2 Event based interaction

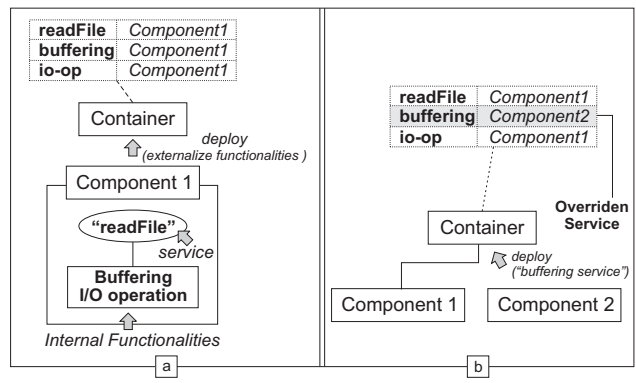
When an event is announced by a given functional component, all the components in the hierarchy of the application that are related to the event must be notified. The interaction based on events is also implemented by containers, and thus there are no direct references among functional components. This process is shown in Figure 3 and the steps are detailed as follows.

1. The component X announces an event named “Event A”.
2. The announcement is directly received by its parent container ($Container 2$), which verifies if any of its child components have to be notified about the event, by inspecting the event table.
3. $Container 2$ forwards the event to the interested components, in this case only the component Y .
4. $Container 2$ then forwards the event to its parent container ($Container 1$).
5. $Container 1$, according to its event table, forwards the event to those interested on it, except the one that announced the event ($Container 2$). Since $Container 1$ is the root of the hierarchy, there is no parent container to forward the event. Thus, the event is only forwarded to $Container 3$.
6. $Container 3$ forwards the event according to its event table that in this case is component K .



As can be seen in Figure 3, there are no references between the component that announced the event (X) and those interested on it (Y and K). Thus, the component that announces the event can be changed without modifying the rest of the structure.

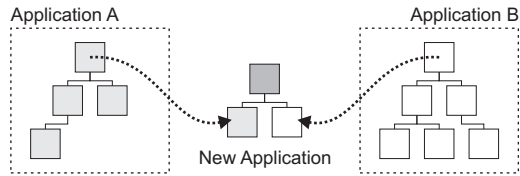
Due to the container-mediated deployment and interaction models, CMS provides mechanisms to override services via “black-box inheritance”. In other words, it is possible to reuse component services without extending the component code, even at runtime. This is possible because a component can require a service implemented by itself. Since the services provided by functional components are accessed via the container, it is only necessary to publish internal component functionalities as external services and access them via a container. Figure 4.a illustrates an example of this process. Consider the “readFile” service, which is implemented by the sequence of functionalities: “buffering” and “IOoperation”. If the internal functionalities are published as services, $Component 1$ can access them via container. Then, the “readFile” service is decoupled of “buffering” and “IOoperation” functionalities.



To override the “buffering” internal functionality, for example, it is only necessary to deploy a component that provides a service with the same name. Figure 4.b presents the service overriding process. In this figure, $Component 2$ overrides the service “buffering” of the $Component 1$. Since there are no references to the $Component 1$, this process, which is based on the *Template Method* and *Strategy* design patterns [6], can be performed at runtime. After deploying the $Component 2$, the “readFile” service of $Component 1$ becomes based on the “buffering” service implemented by the $Component 2$. It is important to note that $Component 2$ does not have to extend $Component 1$, it is only necessary to know the provided and required services.

One can think that a flat architecture could be always better. It could be more interesting in terms of performance and will not require the usage of containers. Also, it will work similarly to registry based systems, event based systems and service oriented architectures, like Jini [13]. However, the main advantage of a hierarchy of containers is that it allows to maintain cohesion of the functionalities provided by their

child components. It makes possible to reuse entire containers without needing to understand the internal components or other containers. Also, it allows recursive composition of applications, since root containers can be viewed as components for other containers. Therefore, an application can be built by integrating containers of various CMS based applications (Figure 5). This feature is not straightforward for architectures based on events, registry or services.

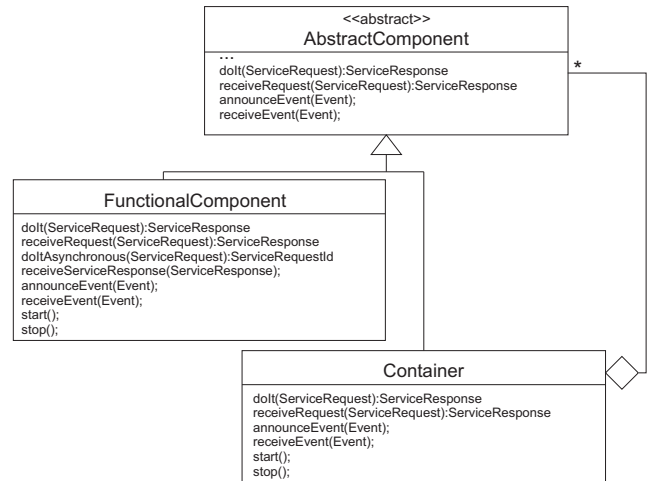


Based on the service and event models, black-box inheritance and recursive composition, the CMS supports all kinds of evolution scenarios: component change, addition, and removal; service and event changes; and architectural changes. In face of unanticipated evolution, an entire application could be dynamically changed. It is only necessary: i) to identify which components will change; ii) to identify which are their dependencies (services and events) and, if necessary, change them; iii) and finally change the identified components. The effort needed to perform this evolution depend on the functional cohesion and complexity of the application. In fact, it could be hard but using the CMS it is always possible.

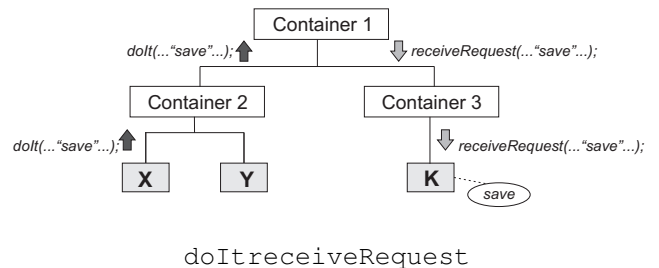
3. Java Component Framework

The JAVA COMPONENT FRAMEWORK (JCF) is a Java implementation of the CMS. The JCF design is based on the *Composite* design pattern [6], which can be applied to hierarchical architectures. Figure 6 shows a simplified version of the JCF class diagram, describing its main methods. Container and FunctionalComponent classes are instantiated in containers and functional components, respectively. The abstract class AbstractComponent insures the recursive composition [6]. Thus, containers are not aware if their children are functional components or other containers. Additionally, it implements the accessor methods. The methods declared in the class AbstractComponent are differently implemented by FunctionalComponent and Container classes, for both the service and the event interaction models.

The service interaction model is implemented through iterative invocations of the `doIt` and `receiveRequest` methods. Such methods are invoked by the components and containers of the hierarchy until the service provider component is located. The function of the `doIt` method is to forward the service request, in a bottom-up way, until reaching the container that contains the reference to the provider. When this occurs, the `receiveRequest` method is invoked, in a top-down



way, until reaching the functional component that implements the service (Figure 7). The syntax for the service invocation methods are `doIt(ServiceRequest)` and `receiveRequest(ServiceRequest)`, where `ServiceRequest` is an object that encapsulates a service name and the parameters needed to execute the service. The result of those methods is a `ServiceResponse`, which encapsulates the service execution result or the exception, if it occurs.



The JCF also implements an asynchronous version of the service based model. The asynchronous interaction implementation is based on the *ActiveObject* design pattern [12]. A component asynchronously invokes a service through the method `doItAsynchronous` and receives a request identifier (`ServiceRequestId`). Then, a new thread is started to request the service through the method `doIt`. When the return from `doIt` occurs, it invokes the method `receiveServiceResponse` for the service requester component, forwarding the service request and the service identifier. Based on the service identifier, the requester component identifies to which invocation the reply refers to.

The implementation of the event based interaction model is based on the *Observer* [6] and *ActiveObject* [12] design patterns. The functionality is implemented through the asynchronous invocation of the method `announceEvent` to announce events to the parent containers (bottom-

up). On the other hand, the invocation of the method `receiveEvent` notifies the events to the interested components (top-down), as occurs with services.

Besides the interaction models specified by the CMS, the JCF implements initialization properties for components. Moreover, JCF provides a mechanism for starting and stopping the execution of the components. The initialization properties are stored in a table for each functional component and can be accessed through the `getInitializationParameter(String)` method, whose argument is the name of the required initialization parameter. The component initialization is implemented by the `start` and `stop` methods. For containers, these methods start/stop all of its components through the invocation of their respective `start` and `stop` methods. For functional components, these methods invoke template methods [6] that are implemented by the component developer, which initialize/interrupt the execution according to the component needs.

4. Application: Wings Pervasive Middleware

Wings is a middleware for pervasive computing that is guided by three issues: context-sensitivity; networking support flexibility; and interoperability both in terms of networking protocol stack and programming language. The basis of the middleware lies on the concepts of *resource*, *context* and *peer*. We define a *resource* as an entity with a description, through which it can be shared, discovered and downloaded, such as an audio file. A *context* encapsulates information about a local peer and the environment on which it is immersed. Finally, *peers* are defined as network nodes having the following set of capabilities: search and sense other peers; share, discover, and download resources, as well as deliver context information.

Due to the sensing capability, Wings has been designed for “infrastructureless” environments. Therefore, the communication between peers must be performed in an ad-hoc way. This characteristic enhances the applicability of Wings in the world of pervasive computing, where the infrastructure is something we cannot always count on. This approach provides the necessary tools to develop applications (*Winglets*) for ad-hoc like pervasive environments such as mobile virtual communities and mobile file sharing. Using Wings, pervasive applications could take advantage of multiple configurations by performing host discovery over different network infrastructures, possibly at the same time. Based on this approach, an application could, for example, discover hosts through *UPnP* (<http://www.upnp.org>), *JXTA* (<http://www.jxta.org>) and *Zeroconf* (<http://www.zeroconf.org>) protocols. This improves the acquisition of context information, since more hosts can be discovered by the applications.

However, it is very difficult to predict which of such protocols will supply the needs of different applications. Moreover, mobile devices are still very limited concerning memory and storage capacities. Therefore, it would not be rea-

sonable to embed in such devices all the existing network infrastructure protocols for each of their wireless interfaces. It becomes necessary a mechanism for inserting and removing such implementations from a device, whenever needed. For that, we use the CMS to encapsulate the peer discovery algorithm and context information mechanisms in software components. Such components, namely Network Infrastructure Components (NICs), may be plugged in and out from the middleware even at runtime.

The CMS successfully provides an effective way of changing *NICs* at runtime, without affecting the rest of the middleware. This is an important feature in a pervasive environment, where the networking protocols involved may change, but users do not want to stop their tasks for replacing one protocol for another. In other words, this process should be performed transparently.

5. Related Work

Different kinds of component models have been proposed. Some examples are *Sun JavaBeans* and *Enterprise Java Beans* (<http://java.sun.com>), and the *CORBA Component Model* (<http://www.omg.org>). Such models have been successfully applied for constructing corporate applications and their middleware implementations provide many interesting services for enterprise software development. However, these models were not conceived to support dynamic unanticipated software evolution. In some cases, their middlewares provide mechanisms and services to perform dynamic changes, but this is not defined in a component model level. The lack of this feature makes difficult the construction of systems supporting unanticipated evolution.

CMS has also some similarities with service oriented architectures: service publishing and provision, transparency of the service provider, flexibility for changes, among others. For instance, Jini [13] is a Java-based technology for the provisioning of services among network nodes. In Jini, services are advertised and discovered in central repositories, like a distributed CMS single (root) container. The work presented in [7] describes a Java middleware for providing services in ad-hoc networks. Such middleware, implemented in Java, is based on a distributed service registry, where each node of the network is able to provide and use services. Other example is the *OpenWings* (<http://www.openwings.org>) framework, which aims at providing service provisioning features targeted to dynamic networks. In these works, dynamic evolution is not supported. In the work of Piccinelli et al [11] the main focus is the dynamic composition of services, and recursive composition of services is also allowed. However, dynamic features are only related to service loading, unanticipated changes are not tackled.

In the context of dynamic evolution models, we highlight HADAS [1]. In HADAS, the focus is the interoperability between distributed components. The reflection concept is used to define the dynamic composition. The developer should define a set of *fixed* behaviors and another set

of *extensible* behaviors for the application. Only *extensible* behaviors can be dynamically composed and changed. Considering that the changes cannot be predicted, the definition of fixed components imposes many difficulties and in some cases makes no sense, since any component can be eventually changed. Another work is Gravity [2] which puts concepts from service and component orientation together for defining a model that supports the adding and removal of components at runtime. For these works, there are no mechanisms to provide recursive composition and the dynamic composition is only allowed for explicitly defined non fixed parts.

6. Concluding Remarks

This paper presented a component model to support dynamic unanticipated software evolution. Such a model, named CMS, provides mechanisms for managing dynamic changes on the software without predicting them, on the fly. A Java implementation of the CMS which allows constructing Java applications supporting unanticipated software evolution was also presented.

Due to space restrictions, some efforts related to this work were omitted. For example, a performance evaluation model has been conceived and code profiling has been performed. Based on performance evaluation results, it can be observed that the hierarchical infrastructure of the CMS architecture does not impact the system performance significantly.

Besides, to make possible a large scale development, an Eclipse-based tool to support the composition activities, called CCT, has been developed. Also, an infrastructure for developing enterprise systems has been built to deal with distribution, security, web, and persistence issues. Persistence is an important feature to allow construct stateful components. Finally, the CMS has been implemented in Python and C++. Multi-language implementations are very important to consolidate the CMS, and also to apply it to different contexts and platforms. The simplicity of the CMS makes possible to implement dynamic evolution without requiring object oriented languages complex mechanisms.

The infrastructure presented in this paper represents a novel engineering support for constructing applications with support to unanticipated evolution. Using CMS, JCF, and CCT, applications can be developed based on reuse, besides improving flexibility and reducing maintenance and evolution time and costs.

References

- [1] I. Ben-Shaul, O. Holder, and B. Lavva. Dynamic Adaptation and Deployment of Distributed Components In Hadas. *IEEE Trans. Softw. Eng.*, 27(9):769–787, 2001.
- [2] H. Cervantes and R. S. Hall. Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model. In *Proceedings of the International Conference on*

- Software Engineering (ICSE)*, page 614623. IEEE Computer Society, May 2004.
- [3] I. Crnkovic. Component-based Software Engineering - New Challenges in Software Development. In *Software Focus*, volume 4, pages 127–133. Wiley, 2001.
- [4] P. Ebraert, Y. Vandewoude, T. D’Hondt, and Y. Berbers. Pitfalls in unanticipated dynamic software evolution. In *Proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE’05)*, 2005.
- [5] M. Fayad, R. Johnson, and D. Schmidt. *Building Application Frameworks*. Wiley, 2000.
- [6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, 1995.
- [7] R. Handorean and G.-C. Roman. Secure Service Provision in Ad Hoc Networks. In *Proc. of the First International Conference on Service Oriented Computing*, volume 2910 of *Lecture Notes in Computer Science*, pages 367–383, Trento, Italy, 2003. Springer Verlag.
- [8] G. Kniesel, J. Noppen, T. Mens, and J. Buckley. First International Workshop on Unanticipated Software Evolution. In *ECOOP2002 Workshop Reader*, volume 2548 of *LNCS*. Springer Verlag, 2002.
- [9] J. Mayer, I. Melzer, and F. Schweiggert. 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. Springer-Verlag, 2003.
- [10] M. P. Papazoglou. Service-Oriented Computing: Concepts, Characteristics and Directions. In *Proc. of Fourth International Conference on Web Information Systems Engineering*, pages 3–12, Rome, Italy, December 2003. IEEE.
- [11] G. Piccinelli, C. Zirpins, and W. Lamersdorf. The FRESCO Framework: An Overview. In *Proceedings of the 2003 Symposium on Applications and the Internet Workshops*, pages 120–123, Orlando, USA, January 2003. IEEE Computer Society.
- [12] J. Vlissides, J. Coplien, and N. Kerth. *Pattern Languages of Program Design 2*. Addison-Wesley, 1996.
- [13] J. Waldo. The Jini Architecture for Network-centric Computing. *Communications of the ACM*, 42(7):76–82, 1999.