Abstract

This paper presents a component model for designing and implementing flexible software components in Java. Our model defines a mapping of how the fundamental concepts of component-based development (CBD) should be implemented using the object-oriented (OO) constructs, available in the Java programming language. The benefit of this mapping is to shorten the distance between a component-based software architecture and its implementation, enhancing the reusability, adaptability and maintainability of component-based software systems.

1. Introduction

One of the main difficulties that software developers face in order to guarantee the conformance of an implementation to a software architecture is the semantical gap between the software architecture description and the programming language used. Usually, software architectures are described with informal or semi-formal artifacts, such as textual descriptions and diagrams. General-purpose architectural description languages (ADL) are still a research area [1]. On the other hand, current programming languages do not incorporate explicitly the key abstractions of software architecture, for instance, the notions of components and connectors [2]. The Java programming language is no exception to this, although it incorporates some abstractions well suited to component-based programming, such as interfaces and packages.

The objective of our work is to design a software component model that enables the implementation of component-based software systems in conformance with its software architecture. Our main contribution is the integration of various concepts and ideas from earlier researches in object-oriented programming into a consistent component-based model. The proposed component model is based on Java and incorporates a set of design guidelines that facilitate the evolution of the implementation while preserving its conformance with the software architecture. Our component model is composed of three inter-operating basic models: the specification model defines the component's provided and required interfaces; the implementation model defines how the services provided by the component are implemented; and the connector model specifies the connections among components. Each basic model is implemented as well-defined patterns, which can be automatically translated into Java source code by means of a tool.

2. Design decisions

Materialization of architectural elements: interfaces and connectors are materialized by a set of infrastructure classes. This allows a direct mapping from the design-time software architecture to the runtime implementation. This mapping contributes to enforce the conformance with the architecture and open the way to the use of dynamic software architectures [3].

Separation of non-functional concerns: components implement functional requirements and connectors implement non-functional requirements, such as distribution, fault tolerance and security. The separation of concerns simplifies the specification and internal design of components and isolates non-functional requirements in connectors. This leads to a more reliable implementation and results in components and connectors with enhanced reusability [4].

Explicit separation between specification and implementation: specification and implementation of components are contained in two distinct Java packages. This physical separation allows the interface’s specifications to be publicly available to the external users, while the component's implementations are restricted to their developers, avoiding undesirable dependencies on the component’s implementations and faulty or malicious access to component's internal state and/or operations [5].

Explicit declaration of Component's dependencies: the component's specification includes provided and required interfaces. Also, a component's implementation shall use only the interfaces defined in its specification. This allows components to be developed and deployed independently, enhancing their reusability [5].
Separation of code inheritance from type inheritance: a concrete class may only inherit code from abstract classes. On the other hand, abstract classes do not implement Java interfaces, which are implemented only by the concrete classes. This simplifies code reuse and guarantees that the external behavior of instantiated objects will be entirely under the control of their concrete classes [7].

Loosely coupling of implementation classes: Java implementation classes interact only by means of Java interfaces. This allows implementation classes to evolve independently, simplifying maintenance and enhancing the robustness of the component’s implementation [7].

3. The Proposed Component Model

This section gives an overall description of the proposed model and its main parts. We show how the architectural components and connectors of a component-based system should be mapped to a Java implementation. Each architectural component is mapped to one package containing two subpackages: the specification package, which describes the external view of the component, and the implementation package, which describes how the component is implemented.

The specification package, on its turn, contains a set of Java interfaces specifying the component's provided and required interfaces. These interfaces have public visibility and comprise all the information needed by the component’s client in order to integrate it in a software configuration and access its services. The implementation package contains the Java class definitions that implement the component's provided interfaces (Figure 1).

Figure 1. Component’s package structure

An architectural connector is mapped to a connector package that implements one or more interface connections. An interface connection connects a component's required interface to one or more component's provided interfaces (Figure 2). Simple interface connections can be implemented with a straight forwarding of service requests to a component's implementation. Otherwise, a connector can implement complex connections, such as persistence, invocation, messaging, and transactions that are largely independent of the interacting component's functionality [8].

Figure 2. Interface connection

3.1. The Specification Model

The specification model specifies the external behavior of a component. It is composed of provided and required interfaces. A provided interface specifies one or more operations that are provided by the component. A required interface specifies one or more operations that must be provided to the component by other components in the software configuration. An architectural component is mapped to a specification package, named `spec`, which is divided in two subpackages: `spec.prov`, containing the provided interfaces provided, and `spec.req` specifying the required interfaces.

Provided interfaces are materialized by façade objects, which are used by connector instances to establish the connections among components. Before a component can be used, its required interfaces must be associated with façade objects, which connect them indirectly to implementation objects. An object of type `IManager` that materializes the component at runtime performs these configuration activities. The `IManager` interface is a mandatory provided interface, similar to COM’s `IUnknown` interface [9]. Like `IUnknown`, `IManager` allows components to be connected. `IManager` defines five operations: `getProvidedInterfaces()` and `getRequiredInterfaces()` retrieve the names of the component's provided and required interfaces, respectively; `setRequiredInterface()` associates a specified required interface to a given façade object, connecting it with a proper implementation; `getRequiredInterface()` returns the façade object currently associated with a specified required interface; and `getProvidedInterface()` returns the façade object that realizes an interface provided by the component.
3.2. The Implementation Model

The implementation model specifies how the operations provided by the component are implemented. Each component's mapping includes a package named `impl` that contains its implementation classes and, optionally, a set of auxiliary interfaces (Figure 3). The implementation classes define the data structures and methods necessary for implementing the component's provided interfaces. An implementation package has four mandatory implementation classes: (i) a `ComponentFactory` class, responsible for instantiating the component; (ii) a `Facade` class that realizes provided interfaces, following the Façade design pattern [10]; (iii) a `Manager` class that realizes the `IManager` interface; and (iv) an `ObjectFactory` class that is responsible for creating instances of the implementation classes.

Optionally, the implementation package may also contain auxiliary interface definitions used internally by the implementation classes of the component. The internal structure of the implementation package is kept hidden from its users. Only the `ComponentFactory` class is visible externally to the package. Also, in order to enhance the maintainability and evolvability of the component's implementation, direct dependencies among its implementation classes are avoided. Once a component is instantiated, the implementation classes interoperate only by means of its auxiliary interfaces. When needed, new objects of the implementation classes are instantiated by the `ObjectFactory` class, following the Abstract Factory and Singleton design patterns [10].

The `ComponentFactory` public class implements a single public static method, called `createInstance()`, that instantiates the `Manager` class, which materializes the component instances. A `Facade` class defines a stable access point for clients of a component's provided interface, decoupling this interface from its implementation class model. Each provided interface of a component should be realized by one `Facade` class, which delegates operations defined for a component's provided interfaces to the appropriate implementation classes. The main responsibility of the `Manager` class is to integrate a component instance into a configuration, maintaining references to the façade objects that realize its provided and required interfaces. It is also responsible for holding the reference of the `ObjectFactory` and passing it to the implementation classes when needed.

3.3. The Connector Model

The connector model specifies the connections among a set of provided and required interfaces, thus enabling two or more components to be connected in a software configuration. A component connector, or connector for short, is a component dedicated to mediate the interactions between other components. A connector can be as complex as a component, however its reusability is compromised by the interfaces that it adapts. Like an architectural component, an architectural connector is implemented as a package. The simple connectors do not define new services interfaces, but uses the interfaces defined by the connecting components. The internal structure of a connector package is similar to a component's implementation package. The mandatory contents of a connector package are the `ComponentFactory` class, the `IManager` public interface, the `Manager` class and a `Facade` class (Figure 4).

Simple connection types, e.g. a call-return between components communicating through identical required and provided interfaces can be implemented directly by a `Facade` class, following the Proxy design pattern [10]. Other more complex connections, e.g. between mismatching remote interfaces may require additional implementation classes, auxiliary interfaces and an `ObjectFactory` class. Connectors can also provide services, such as persistence, invocation, messaging, and transactions that are largely independent of the interacting component's functionality.
4. Experimental Works

We are currently experimenting our model to develop two real-world software systems. The first experiment consists of a component-based development of an application in Bioinformatics, which was executed by a company's software team as part of an effort to introduce an architecture-centric software engineering approach. Preliminary results of this first experiment have shown that the model is feasible in practice, and reduces time-to-market. A simpler version of the model was used by the company, which has chosen not apply the connector model. The second experiment is a component-based development of a Web-based traveler information system for a highway department. A team of graduate students based on a real-world specification is implementing this project, and it is part of a broader research on fault-tolerant software architectures. With this experiment, we have learnt that our model can be applied in various levels of sophistication, that is, one can choose to start using a reduced version of the model and to improve the model according to the system's evolution requirements.

5. Conclusions

This paper presents a component model for the implementation of component-based software architectures in Java. The architectural abstractions, such as components, interfaces and connections, are realized in the system implementation using constructs available in the Java language. With this, the distance between the software architecture and the system implementation is shortened, contributing to guarantee the architectural conformance of the implementation. The proposed component model, based on a set of fundamental design patterns and guidelines, also enhances the reusability, adaptability and maintainability of the developed systems. The main contributions of our work are the integration of various concepts and ideas from earlier researches in software architecture, component-based development (CBD) and object orientation into a consistent Java component model.

The apparent complexity of the proposed model, which needed more elements than we expected beforehand, reflects the distance between the architectural abstractions that are essential to CBD and the Java language. Unlike, JavaBeans [11] and ArchJava [12], our approach does not define a higher-level language to describe architectural configurations. We define a pure Java component model at the source code level. In this sense, our model is similar to the JavaBeans model [13]. Like JavaBeans, we use patterns to implement components in Java. However, in our model, a component has a higher granularity than a JavaBeans component (Bean), since we define a set of mandatory infrastructure classes to implement a component, while in JavaBeans a single class is considered a Bean. Also, the notion of required interfaces and connectors is not present in JavaBeans, where the interaction among components is realized by events. In relation to COM model, there is not also the required interface and the component instance concepts, and each COM's interface should inherit from the IUnknown interface, which defines basic infrastructure services, while our model has a mandatory provided interface IManager, which materializes a component instance at runtime and defines infrastructure services.

Currently we are initiating an effort to automate the mapping from a software architecture description to a Java implementation using the proposed model as a template. The objective is to integrate this automatic code generation into a tool with support for the xADL architecture description language [1]. We choose xADL because it is defined as a set of XML schemas, which provide basic support from the many available commercial XML tools and it allows the definition of extensions which can be customized to our component model.

Acknowledgments

M. Silva Jr. and C. Rubira are supported by CNPq/Brazil. P. Guerra is supported by CAPES/Brazil.

References