STANDARDIZING COMPONENT INTERACTION PATTERNS IN PRODUCT-LINE ARCHITECTURES

Bo Nørregaard Jørgensen, Wouter Joosen

The Maersk Mc-Kinney Moller Institute for Production Technology, Odense University, DK-5230 Odense M, Denmark.
{bnj, wouter}@mip.sdu.dk

0. Abstract

In this position paper we describe a set of interaction patterns that have been used to facilitate the composition of components in the product-line architecture of a robot control system. The primary purpose of standardizing component interaction is to reduce the learning curve of the inner working of the product-line architecture. Secondary it results in increased readability of the architecture diagrams.

1. Introduction

To facilitate the composition of components in product-line architectures the applied component framework must provide guidelines for the interaction of components. Guidelines prevent interface and contract mismatch between components developed by different component developers. This is important since total anarchy will require very skilled application developers that are able to write suitable adapters for gluing the components together. If too much time is spend on gluing components together instead of composing functionality then the promised benefits of component-oriented software development is lost. Basically a good component framework should ease the development process for both the component and the application developer, hence the guidelines for interaction must capture any reoccurring pattern for sound interaction between components.

Guidelines for component interaction can be enforced by requiring components to use a set of predefined interaction patterns. A set of interaction patterns can be defined as a number of syntactical conventions for interface and method names. One well-known component-framework that applies this approach is the Java Beans framework [1]. However the interaction patterns in the Java Beans framework is aimed at reactive applications and are therefore not sufficient for industrial control application where periodic activities are common. At least, interaction patterns for periodic activities must be defined in order to complete the picture.

The paper is organized as follows: in section 2 we discuss five interaction patterns and how they relate to the temporal behaviour of the components' information processing. Guidelines for applying the presented interaction patterns are given in section 3. Section 4 describes the application of interaction patterns in the product-line architecture of a robot control system. Related work is discussed in
section 5. Directions for future work related to component interaction are discussed in section 6. Finally we conclude in section 7.

2. Interaction Patterns

In this section we discuss five interaction patterns. The patterns have their origin in the temporal behaviour of component interaction, that is, how components interact throughout time. This includes both composition-time and run-time. At composition-time component interaction comprise the modification of component properties and the creation of connections between components. This activity involves the components of the development environment and the application. During runtime, component interaction is only concerned with the components within the application. Run-time interaction can be sporadic or periodic and utilize either push or pull invocation styles. For instance, event notification in reactive systems is a classical example of sporadic push invocation.

The first interaction pattern that we discuss is identical to the Property design pattern in the Java Beans specification. It is included here for completeness of the discussion. The second one is a modification of Java Beans’ Event design pattern. The third one handles periodic activities with isochronous nature. The fourth and the fifth facilitate the request of well-defined services offered by other components.

2.1 Property

The appearance and behaviour of a component can be customized through its properties. For instance, a component that offers multiple algorithms for the same computation can dedicate a property to select the algorithm to use. An algorithm can then be selected through the property’s setter method by an application developer at composition time or by another component at runtime. In general properties can be modified and accessed programmatically by other components by calling their setter and getter methods. This allows modification of properties at both composition-time and run-time. The used of getter and setter methods gives full programmatically control of a component properties.

Participants

Owner: The component that the property belongs to.

A component must use the syntactical conventions shown in Fig. 1 for its getter and setter methods.

```java
public void set<Property>({PropertyType} property);  
public <PropertyType> get<Property>();
```

5 The Java Beans specification refer to the use of syntactical conventions as design patterns.
Fig. 1 Methods for modifying and accessing properties

Modifier: The component that accesses or modifies the property.

No additional interface convention is needed for the modifier. The modifier uses the target component's getter and setter methods to modify the property.

If both the getter and setter methods are available the property will be read-write. If only the getter method is available the property will be read-only. In case only the setter method is available the property will be write-only.

Applicability

Use the Property pattern to configure components at composition-time or to change the properties of components during run-time.

Temporal behaviour

Properties need not just to be simple data fields or object references they can be computed values based on other properties. Updating a property may have various programmatically effects depending on whether the property's updating mechanism uses eager or lazy evaluation either the property's setter or getter method can result in the firing of property change events or re-computation of the property value.

Benefits and limitations

The properties which control the customization of a component is made explicit to the application developer.

2.2 Event

The Event pattern fits applications with a reactive nature well. Such applications often consist of a collection of loosely coupled components in which each component carries out some operation and may in the process trigger operations in other components by firing events. By convention an operation that is triggered by an event do not return a result. A component may fire events spontaneously as a result of a change in its state or as a change in an environment parameter that it is responsible of monitoring.

Participants

Event Listener: A component that needs to be notified when some event occurs.

An event listener must implement the interface that specifies the method to invoke when the event occurs. Fig. 2 shows the syntactical convention for defining an event listener interface.

```java
public interface <Event>Listener {
```
public void on<Event>{<Event>Event event};
}

Fig. 2 Interface of an event listener

The event source uses the method on<Event> to notify the event listener about the occurrence of an event.

Event Source: The component that fires the event.

An event source provides methods for registering and de-registering event listeners:

public interface <Event>Source {
    public void add<EventListenerType> (<EventListenerType> listener);
    public void remove<EventListenerType> (<EventListenerType> listener);
}

Fig. 3 Registration methods of the event source

Invoking the method add<EventListenerType> adds the given listener to the set of event listeners registered for events associated with the <EventListenerType>. Similarly invoking the method remove<EventListenerType> removes the given listener from the set of event listeners registered for events associated with the <EventListenerType>.

Applicability

Use the Event pattern when a component have to inform other components about changes that have their interest.

Temporal behaviour

Events are generated sporadic at a source and propagated to a sink (a listener). The generation of events can be instantaneous or it can require a substantial time period. A simple transformation of an I/O interrupt from an external device is an example of an instantaneous generated event. Events that are fired as a result of re-computing a component’s properties are examples of time consuming event generation. The source does not impose any temporal constraints on the listener’s processing of an event. Such constraints must be associated with the event at the listener side, that is, the acceptable response time for a particular event should be specified at the listener side.

Benefits and limitations

Provides loosely coupling of listeners to the event source, this relief the source from knowing about the actual type of the individual listeners.
2.3 Publisher

The Publisher pattern is suitable for systems with isochronous nature. Isochronous pertains to the property of being equal or uniform in time. In information technology, this may refer to both processes and data sequences. Processes are said to have isochronous nature if they generate or process regular amounts of data at fixed time intervals. Similarly, data sequences are said to be isochronous if their elements are equidistant in time and require processing at equal time intervals.

Sampling is the most common source to data sequences of isochronous nature. Other sources are computer generated data sequences such as computer animation, and digital music. Audio and video are typical examples of data sequences with isochronous nature.

Participants

Subscriber: Component that consumes data.

```java
public interface <Data>Subscriber {
    public void on<Data>Ready(<DataType> data);
}
```

Fig. 4 Interface of a subscriber

The publisher uses the method on<Data>Ready to deliver data at the subscriber.

Publisher: A component that generates data that other components use to perform their function.

```java
public interface <Data>Publisher {
    public void setPeriod(long period);
    public void add<DataSubscriberType> (<DataSubscriberType> subscriber, ...);
    public void remove<DataSubscriberType> (<DataSubscriberType> subscriber);
}
```

Fig. 5 Registration methods of the publisher

A component that consumes data register itself by invoking the method add<DataSubscriberType>. This method adds the component to the publisher's set of subscribers. Similarly, invoking the method remove<Data SubscriberType> removes a subscriber from the set of registered subscribers. The registering method can be designed to take additional arguments. Those arguments can be used to describe the quality of service level that a subscriber requires. The time period between publication of data is set by the method setPeriod.

Applicability

Use the Publisher pattern when a component has to deliver data to other components at fixed time intervals (periods).
Temporal behaviour

In the Publisher pattern the data processing at the subscriber and the publisher side is mutual dependent. The subscriber must be able to process the data published by the publisher at the same rate as the publisher produces it. This implies that the time required to process the published data, must be less than the publication period. The subscriber is responsible for verifying this relationship. In case of a timing mismatch the subscriber should retain from subscription. Similarly, the time required to produce the data must be less than the publication period.

The publication period can either be a property of the published data or has its root in the requirements of the subscriber.

Benefits and limitations

If more components subscribe to the same publisher component they have to share the same update period. This is not always practical or possible and thus may introduce the need for more publishers of the same kind within a system (i.e. more publishers which publish the same data but with different periods). Sensing is one application area where this may be the case.

2.4 Synchronous Service Provider

The Service Provider pattern captures the typical Client – Server situation where one component needs another component to perform a service on its behalf. For instance the computation of some value. A service represents some third part functionality that the component depends on. The component that offers the service contains the necessary means to perform the service, meaning that the component does not depend on the availability of third party components for providing the service. Any such component will be packaged with the component.

Participants

Requestor: The component requesting a service from another component.

Since the result of the service is returned as part of the invocation, no syntactical conventions are necessary for returning the result. The requestor is implicit given by the synchronous invocation mechanism. However we need a method for setting the service provider, see Fig. 7.

```java
public interface <Service>Requestor {
    public void set<Service>Provider (<ServiceProviderType> provider);
}
```

Fig. 6 Interface of service requestor

Provider: The component providing the service.

The provider must follow the interface specification convention shown in Fig. 7.
public interface <Service>Provider {
    public <ServiceResultType> perform(<Service>DataType data);
}

Fig. 7 Interface of service provider

Invoking the method perform(<Service> with an argument of type <ServiceDataObject> implies that the service will be performed on the data contained in the <ServiceDataObject> object. The result will be returned in the <ServiceResultType> object.

Applicability

Use the Synchronous Service Provider pattern when a component has to wait for the result of a service provided by another component in order to perform one of its own services.

Temporal behaviour

The service offered by a component must be performed within the temporal constraints of the requestor. If this is not possible the service should not be requested. This is important since the requestor waits until the provider has finished its execution. A late reply can cause a violation of the temporal constraints imposed on the requestor by the application environment. This implies that the temporal properties of a service should become part of the component's specification.

Benefits and limitations

The method set<Service>Provider makes the coupling between a component and its requested services explicit. This emphasizes the component's dependencies to other components. The use of the Service Provider pattern makes the component requesting the service independent of actual type of the component implementing the service.

2.5 Asynchronous Service Provider

The asynchronous version of the Service pattern extends the synchronous one by defining a callback interface for returning the result to the requestor. This allows the requestor to perform other tasks while waiting for the result. We do not consider the case where no result is returned since this kind of behaviour is covered by the Event pattern.

Participants

Requestor: The component requesting a service from another component.
public void set<Service>Provider
    (<ServiceProviderType> provider);
public void on<Service>Response
    (<ServiceResultType> result);

Fig. 8 Interface of the service requestor

Provider: The component that provides the service.

gap: Fig. 9 Interface of the service provider

Applicability

Use the Asynchronous Service Provider pattern when a component can perform other actions while waiting for the result of a service provided by another component that it needs to perform its own service.

Temporal behaviour

As in the synchronous case the service provider must deliver the result of the service request within the temporal constraints of the requester.

Benefits and limitations

Increases concurrency within an application.

3. When to Use Which Interaction Pattern

The choice of interaction patterns for interconnecting components in product-line architectures should be governed by the implicated components' need for interchanging information. These needs are closely related to the information processing done by the components. Here information processing pertains to the activities of property modification, state change notification, data production and service request. For instance, the Event pattern is a natural choice if the information processing done by a component leads to sporadic changes that other components should be notified about. Similarly, if the exchange of information between two components occur periodically the Publisher pattern should be applied. A component that need the service of a third party component to deliver its own service can be connected to this component by using the Service Provider pattern.

4. Applying Interaction Patterns

In this section we describe the application of the discussed interaction patterns in a recent developed component-oriented software-architecture for a robot control
system. The robot control system is developed in a research project that address control of high-speed robots [2]. The robot control system aims at the application area of spray painting. However a requirement to the developed architecture is that it must be applicable to previously explored application areas such as robot welding and cutting. This implies that the architecture must be adaptable to different robot types and control strategies.

4.1. Component Architecture

Inspection of previous applications for robot welding and cutting leads to the identification of the components showed in Fig. 10. The functionality of the components relates to the two logical tasks of robot motion 1) deciding how to move the robot to the target position and 2) moving the robot to the target position.

Fig. 10 Components of the robot control system

Fig. 10 shows the use of interaction patterns to direct the interaction between the components in the robot control system. The use of syntactical conventions allows us to deduce the behaviour of the individual components from the interfaces that they implement. As an example let us take the TrajectoryPlanner component. From its TargetChangeListener interface we know that it listens for occasional changes in the target position and from its PositionPublisher interface we know that it periodically has to publish new positions. From this we can deduce that the TrajectoryPlanner component must generate intermediate positions between target positions since the target only changes occasionally and the TrajectoryPlanner component publishes new positions periodically.

Briefly the individual components are responsible of performing the following tasks:

Task Scheduler: handles execution planning of robot tasks. For example welding of a line segment between to iron plates.
Trajectory Planner: responsible for planning a collision free trajectory for the robot when executing a task.

Parameter Optimizer: evaluates and updates the robot model accordingly to the observed physical behaviour of the robot.

Force Generator: computes the forces that have to be applied to the robot's joints in order to obtain the desired motion.

Joint Controller: responsible of applying the generated forces to the robot's joint actuators.

Physical Model: contains the mathematical description of the robot's physical properties (e.g. friction, momentum, weight, etc.).

4.2. Component Interaction

The component architecture applies the Publisher pattern to connect components that produce and consume isochronous information. For each information-producing component the production period depends on entities in the application domain (i.e. the type of robot, task, and control strategy). Analogously, the Event pattern handles the exchange of sporadic changing information. The change of parameter values in the physical model component is one example of sporadic changing information. The Property pattern is utilized to facilitate the configuration of components. For instance the applied force control strategy is selected through the Force Generator component's control strategy property. This can be done at both composition-time and run-time. Finally the composition of application specific components from other more general-purpose components is done by the synchronous Service Provider pattern.

At the moment we have not found any need for applying the asynchronous Service Provider pattern in the component architecture, but this may change as the architecture evolves.

5. Related Work

Other development and research efforts have addressed aspects of interaction between independent developed components [1,3]. Where these efforts are concerned with the description of the dynamic behaviour of component interaction (i.e. specification of protocols) our work focuses on the temporal behaviour of component interaction.

In [3] the concept of a Connector is proposed for describing the interconnection between components. A Connector is defined as a set of roles and a glue specification. The roles describe the expected local behaviour of each of the interacting parties. The glue specification describes how the activities of the roles are coordinated. This approach allows one to define reusable connector classes that
can be instantiated as needed to interconnect the high-level computational components of the system.

The JavaBeans [1] component framework uses simple design patterns consisting of conventional names and type signatures for methods and interfaces to specify what properties, events, and public methods a JavaBeans component supports. The use of design patterns allows the application developer to quickly identify the particular events and services of a component. However JavaBeans' component model only address event driven component interaction which is not enough for industrial control application where exchange of isochronous data occur frequently.

6. Future Work

Future experiments will investigate how the presented set of interaction patterns can be used to deduce information about a component's resource requirements. Resource requirements include a component's memory, CPU and network consumption. Focus on this issue is fundamental due to the strict temporal requirements that are imposed on any software system operating in an industrial setting. Without some form of resource awareness temporal overload of system resources such as CPU, memory and network bandwidth leads to processing delays that can cause parts of the system to fail. In a worst-case scenario the failure of a single part of the system may bring the whole system down. This happens if the system has no means to discover and protect itself against resource overloading [4]. To the best of our knowledge there is no component framework available today that addresses this issue.

7. Conclusion

From the software engineering perspective a key goal in the robot controller project is to identify and document a set of interaction patterns that can reduce the time required for new component developers to learn the component architecture of the robot control system. This goal has so far been achieved by requiring the component developers to follow a set of syntactical conventions that enable other developers to easily recognize the interaction patterns and temporal behaviour of the components. The documented set of interaction patterns comprises a framework for 'wiring' components together.

References

